Software Verification for Programmable Logic Controllers

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Ralf Huuck

Kiel,
2003

1. Gutachter	Prof. Dr. Yassine Lakhnech
2. Gutachter	Prof. Dr. Willem-Paul de Roever
3. Gutachter	Dr. habil. Oded Maler
4. Gutachter	Prof. Dr. Thomas Wilke
Datum der mündlichen Prüfung	17. April 2003

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Computers are everywhere. As a matter of fact the routine of our daily lives is more and more supported by, and at the same time dependent on, computer driven hardware. Most of the electrical appliances we use at home, e.g., the TV set, the refrigerator, and the telephone are nowadays driven by micro-controllers or computers in general. And even computers take over the task to stabilize cars during driving or to ignite air-bags; they restrict access to certain areas or secure bank transfers, and so on. But a vast part of these systems we never see. All the things natural to us like water, food, and electricity rely on computers as far as their distribution and manufacturing process is concerned.

Wherever we get directly or indirectly in touch with computers and computer driven hardware, their main purpose is: control. The control of the refrigerator temperature, the traction of the car wheels, the scheduling and transport along the assembly lines that can food, and the control of many safety-critical applications such as fission control in nuclear power plants.

A question that arises is, what kind of systems are used for all these controlling processes? Mostly not general-purpose PCs but specific industrial computers. The reasons are simple and stem from the requirements of such control systems: they have to be robust, reliable and cheap. Robustness to ensure that these systems work also under unexpected conditions, which might be heat, dust, and electro-magnetic noise [Mor]. Reliability in order to have them running 24 hours a day over 5 or even 10 years. And they have to be cheap to enter a mass market and to reduce the costs of the manufacturing process.

Next to micro controllers, specific purpose computers on a single chip, one prominent class of industrial controllers are so called *programmable logic controllers* (PLC). First developed during the early 1970s, PLCs started as

simple devices to replace electro-mechanical relays. Using integrated circuit technology, they performed simple sequential control tasks, in isolation from other control and monitoring equipment. These simple devices have grown into complex systems capable of almost any type of control application, including motion control, data manipulation, and advanced computing functions. Nowadays, PLCs are extensively used in the field of automation and they are integrated into much larger environments, requiring communication with other controllers or computer equipment performing plant management functions [Lew98] [BMS99].

Control systems driven by PLCs are often complex, safety critical, and involve a lot of money. Any failure of these systems might not only result in a significant financial loss but lead to casualties as well. Hence, next to robustness and reliability, their actual programming and the correctness of the programs plays a vital role.

Since PLCs have been for a long time subject to the engineering community only, their programming languages evolved out of historical needs. This means, that they are particular suitable to implement controlling tasks as the solution to control theoretic problems. Therefore, the used programming languages are often close to relay ladder logic or assembly language.

In contrast to computer science, where a number of new programming languages evolved, driven by the latest theoretical results, industrial programs often follow the well-known and established concepts of the first programming languages. This means, that high-level languages including object-orientation, abstract data-types, and graphical programming environments, which support a more abstract and problem-oriented approach, have not or to a much lesser extent found their way into industrial and PLC programming [Mal99]. In this sense industrial programs are much more exposed to design errors and more likely to fail than state-of-the-art programming languages which demand a strict programming regime and foster a more abstract development process.

There are at least two ways to improve the current situation.

1. One way is to design new programming languages and environments for PLC programming. These languages should be high-level to allow a development on an abstract, problem-oriented level and at the same time they should be simple and rigorous enough to enable a clear understanding and to support the development of safe programs right from the beginning.

2. Another approach is to keep the existing programming languages and to develop analysis and verification methods for PLC programs. These methods should then aim at proving the correctness of PLC programs and detecting even subtle bugs.

In this work we follow the second approach. The reasons for this are

simple: today there are many PLC programs in existence which will not be rewritten; however, there is a chance that they might be re-analyzed. Moreover, it is not likely that people change domain specific languages easily since there is a long tradition in using them and it takes time to build confidence in new languages. On the other hand, a clear understanding of the semantics of the current languages and tools to analyze the developed programs is much more likely to make an impact on PLC programming.

The goal of the remainder of this chapter is to show how software engineering can support the development of correct PLC (and in a broader sense industrial) programs and which techniques we intend to develop and apply in this work to achieve the correctness. As a prerequisite to understand the needs and developments for control system we give a brief introduction to control systems engineering and control theory in Section 1.2. Moreover, we discuss in Section 1.3 how controlling issues are theoretically covered in computer science. From this perspective we take a look at PLCs and their programming languages in Section 1.4. In Section 1.5 we outline the major challenges for the verification of PLC software and our contribution to master these. Section 1.6 outlines the remainder of this thesis, while Section 1.7 gives references to work we published prior to this thesis on the same topic.

1.2 Control Engineering and Control Theory

Before taking a look at control system engineering and control theory we briefly explain a number of terms. A *control system* is defined by Nise [Nis00] as follows:

> A control system provides an output or response for a given input or stimulus.

This definition is vague but it stresses the importance of an input/output behavior. What actually should be achieved is a *desired response* to a given control problem, which is defined as the idealized instantaneous behavior that we would like from the system. For instance, the temperature in a refrigerator should always be between 4 and 6 degrees Celsius. Of course, before connecting the refrigerator this requirement is in general not satisfied. Once connected, it starts (most likely) to cool down; this phase is called *transient response*, i.e., the gradual change in the system as it approaches its approximation of the desired response. Once the system has finished changing (cooling down), its approximation of the desired response is called the *steady-state response.* E.g., the fridge is cooled down to 5 degrees Celsius. Commonly it is required that once a system has reached its steady-state response it can maintain it, i.e., it is *stable.* This means the temperature will remain within 4 to 6 degrees and neither completely leave this range nor oscillate around this range. Additionally, *robustness* is often required. This

implies that even if the systems parameters change it should still function correctly. E.g., the fridge control should be unaffected as much as possible by the number of items and their temperature inside it.

One of the first incidents in industrial engineering that suggested the need for control systems engineering took place in the beginning of the nineteenth century. Watt built his first governor for a steam engine, a device to control the speed of the engine. At that time the engines had such a poor response and such a friction that the engine was always stable, even though the transient response was slow. However, as steam engine technology improved, it turned out that the governors that worked before did not do so anymore, since their transient respond was simply too slow which more often than not led to a blow up of the engine. From this time onwards control engineering became a science and contributed a lot to the development of ship steering, robotics, air-craft and missile control, just to mention few.

Control theory concerns the mathematical formalization of problems in control, the desired properties as stated above, and the control systems that are the required solutions to the problems. The modeling is traditionally close to the physical components involved. This means, e.g., that for mechanics and mechanical networks parameters such as force, velocity and mass are taken into account. For electrical system elements the models include, e.g., the current, voltage, and torque.

The implementation of control systems started off with analog boards and (electro-)mechanical systems reflecting well the continuous control theory. At some point analog controllers were replaced by digital ones. An advantage is that digital systems were able to replace the relays used in electrical systems. Relays were notorious for wearing out after a short time, while in digital systems switching from one mode to another is all implemented in software and, therefore, free of mechanical wear. The distinct characteristics of digital systems compared to their analog counterparts is their *cyclic* behavior: periodically, inputs are sampled, computations based on the inputs and the internal state of the controller are performed and, afterwards, the corresponding output is emitted. These systems match well the theory of sampled control and signal processing systems [CF95].

Often the programming languages for these digital systems, like PLCs, are still close to descriptions of their electro-mechanical counterparts which they replaced. This has, e.g., as a consequence that a description language for relay ladder logic is used as a programming language for PLCs. For the same reason assembly languages, introduced as low level programming languages for digital systems, are as well part of the widely used PLC programming languages.

With history in mind, it is clear why PLC programming languages tend to be arcane and low-level compared to current developments in software engineering. They are the result of an evolution from electrical to digital systems, and often remained as they were first designed because this proved

to be practical.

In contrast, many programming languages developed by computer scientists were designed from scratch driven by latest theoretical insights without the urge to take care of historical needs and evolution.

1.3 Control Systems in Computer Science

Computer science often tends to look at physical systems from a more abstract level when compared with control theory. The fundamental issues seem to be of higher interest than an actual model, which is close to the underlying physics. We like to mention two aspects that reflect the ideas of control systems: The continuous/discrete world and the input/output behavior.

The view of control system as mixed continuous discrete systems led to the notion of *hybrid systems* [NSY93] [ACHH93]. Research, both in the computer science and the engineering community, has focused on hybrid systems for many years in order to develop appropriate system models and verification methods [Mal97, iee98, aut99, NK00].

Achievements have been made in describing such systems, to visualize their effects, and to simulate some behaviors [mat]. On the verification side the development of appropriate system models led to proof systems which allow the manual or computer supported verification of system properties, and sub-classes have been detected which enable the use of automatic verification tools to check certain kind of properties [ACH+95, DM98, AHLP00, DN00].

Overall, the verification of hybrid systems is still complex, time and memory consuming and is still far from any automatic approach. Such an approach, however, is limited by decidability results [HKPV98], and this means that there are fundamental limits to handling the general class of hybrid systems in a fully automated manner.

Another characteristic of control systems is their input/output behavior. The continuous interaction of control systems with their environment at the same speed as the environment led to the notion of *reactive systems* [HP85, BG92]. Characteristic for reactive system is their instantaneous (or negligibly short) reaction time to *events* stimulated by their environment. Every reaction of a system produces a response to the environment quickly enough not to miss any new event such that there is no overlap in reactions. This assumption is the basis of the *synchrony hypothesis* [BG92]:

> The reaction of a system takes negligible time with respect to its environment.

The synchrony hypothesis is also the basis for a number of so-called *synchronous languages*. These languages include state oriented imperative synchronous languages such as Esterel [BG92], StateCharts [Har87], Grafcet

[DA92], and Argos [Mar92], as well as data-flow languages such as Lustre [HCRP91] and Signal [BLJ91]. Although these languages found their way into industry, e.g., telecommunication and avionics, they are not used for PLC programming and are widely unknown in the respective industrial fields.

Note on a side remark that the synchrony hypothesis is a very strong hypothesis that does not always hold in practice. Moreover, the synchronous approach has to deal with several other obstacles when it comes to distributivity and robustness in practice. An overview is given in [Cas01]. We do not intend to go into detail here.

1.4 PLCs and PLC Programming Languages

The hardware of a PLC (see Figure 1.1) consists of a microprocessor based CPU, a memory, input and output ports where signals can be received, e.g., from switches and sensors, and sent to actuators, e.g., to motors or valves. A PLC is equipped with an operating system that allows to load and run programs and perform self-checks. Traditionally, the PLC operating system must respond to interrupts and must be real-time. Programs for PLCs are developed and compiled on external devices and downloaded to them afterwards.

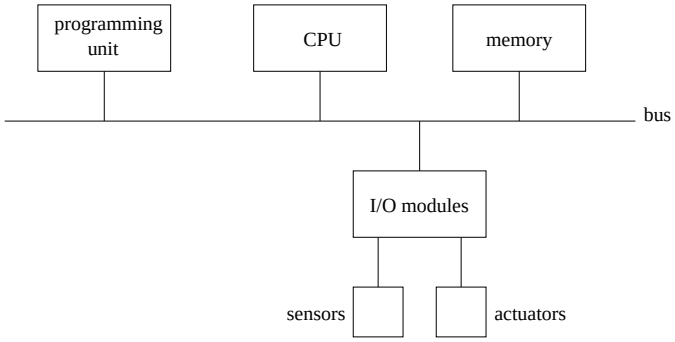

Figure 1.1: PLC architecture

The main difference to conventional systems such as PCs is the cyclic operation mode illustrated in Figure 1.2: PLC programs are executed in a permanent loop as mentioned in Section 1.2. Although PCs have also input/output functionality to exchange information with, e.g., a keyboard or a mouse, this i/o functionality is not their main reason of existence.

From a computer science perspective, PLCs have the characteristics of real-time, reactive systems. If their environment is taken into account they resemble hybrid systems.

There exist different programming languages for PLCs. They have been designed with an emphasis to implement control tasks, each intended for

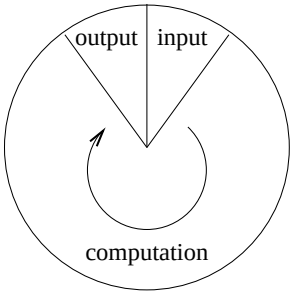

Figure 1.2: PLC cyclic execution

a specific application domain, and based on the background of the control engineers who use them. In order to achieve more conformity of the different PLC programming languages the standard IEC 61131-3 was developed.

Instruction List (IL) is a low level assembly language with one register called *current result.* IL is still one of the main programming languages for PLCs used in Europe, since it allows compact code and is very close to hardware programming.

Structured Text (ST) is a PASCAL-like language. It is a higher level language than IL and provides more convenient structuring and organizing constructs such as while-loops, if-then-else-conditionals and so on. However, ST is used to a much less extent than IL.

Moreover, three graphical languages are defined:

Ladder Diagrams (LD) sometimes also called *Relay Ladder Logic.* Indeed, LD programs resemble relay logic. The diagrams are drawn in a graphical editor, e.g., on a PC, and then compiled and moved to the PLC. Like IL the LD programming language is very low level, but serves its purpose whenever relay ladder logics have to be compiled into software. Until today LD programs are the major programming language for PLCs used in the U.S.

Function Blocks (FB) are essentially a data-flow language for describing a network of functions connected by signals. Pre-defined functions, generally drawn as boxes, are combined by connecting inputs and outputs in a desired manner. They are interpreted along their flow of control, i.e., function blocks are interpreted along their connecting paths and provide a better overview and understanding of how functions relate than, e.g., LD programs.

Sequential Function Charts (SFC) is a graphical, high-level programming language for PLCs which aims at providing a clear understanding

of the possibly interwoven program parts. SFCs allow to decompose and structure program parts and include interesting concepts such as parallelism, activity manipulation and hierarchy. Historically, they take over ideas from *Petri nets* and *Grafcet* [DA92].

Most of these programming languages have additional PLC specific features and allow the use of timers and to access system time. It is to be expected that in several application areas PLCs will be replaced by PCs in the future. However, the main principles such as a cyclic operation mode and the use of timers will be maintained. Also it is unlikely that there will be a change of programming languages in the near future. One indication for this is that although higher-level languages such as ST are standardized, still the lower-level languages such as IL and LD are most commonly used. One exception to this are SFCs.

1.5 Scope of this Thesis

The goal of this work is to develop and apply software analysis techniques for PLCs which lead to more reliable PLC programs. In this section we outline the challenges which should be faced, as well as the solutions to them developed in this thesis.

1.5.1 Major Challenges

Although PLC programming languages are standardized in IEC 611321-3, no formal semantics is given there; this obstructs formal reasoning. Moreover, the informal descriptions presented in the standard are often too incomplete and ambiguous to directly infer a formal semantics.

The different PLC languages are of different nature. They range from the low-level IL languages to the high-level graphical language SFCs with its many unique features. Their different nature implies different analysis objectives and, therefore, the use of different analysis techniques.

These analysis techniques should be applicable in an industrial setting. Hence, they should scale to real-life problems and also be applicable by non-experts in formal verification.

1.5.2 Contribution of this Thesis

Throughout this work we focus on two PLC programming languages: IL and SFCs. These two languages complement each other in their level of abstraction, their general size in terms of lines of code, and above all their programming features. Therefore, they serve as a basis to counter the various aspects of the challenges presented above.

Development of a Formal Semantics for SFCs and IL

A prerequisite for any formal reasoning about system behavior is a formal semantics, i.e., a mathematical framework that represents the behavior of such a system.

For SFCs, we first investigate their informal semantics as outlined in IEC 61131-3 and point to a number of ambiguities and incomplete descriptions in the standard. We discuss several possibilities to remedy the situation and, based on this, present a unifying formal operational semantics based on transition systems. The semantics is unifying since it can be adjusted by parameters to different interpretations of the standard and, hence, also to different existing implementations. The semantics models the cyclic behavior and is comprehensive, i.e., it covers all the main SFC features such as hierarchy, parallelism, priorities as well as actions and action qualifiers. An extension to timed SFCs is sketched.

Although the IL semantics as described in the standard is less ambiguous, a formal approach does not exist. We develop a *structural operational semantics* [Plo81] for IL programs. This semantics covers a significant core set of the IL language and respects the cyclic behavior inherent to PLCs.

Combining Algorithmic Analysis Techniques

Due to the different nature of the examined languages we develop different verification techniques for each of them. Common to all techniques is that they are *algorithmic* and need no or little interaction with a user. Hence, they are mostly applicable by non-experts.

We propose a model checking [QS82][CE82] approach for SFCs. We introduce a finite abstraction for SFCs and develop a translation from such SFCs to the input language of a model checker. This enables the analysis of SFCs for a rich class of properties described in temporal logics [Pnu81]. Moreover, we propose a method also based on model checking to determine *safe* SFCs. These are SFCs that comply to certain structural requirements.

IL programs make use of potentially infinite data structures such as integers and often consist of hundreds or thousands of lines of code. Therefore, we propose verification techniques that are particularly scalable and can approximate infinite system behaviors. We choose and develop combinations of abstract interpretation [Cou78][CC79] and data flow analysis [Hec77]. Abstract interpretation is used to approximate program behavior, i.e., to determine the potential range of program variables during program execution. Data flow analysis is used to analyze the information flow and check, e.g., for unreachable or dead code. The combination of both techniques gives a significant boost to the obtained analysis result.

Furthermore, we develop an abstract simulation for testing IL programs with sets of input values simultaneously. We also present a heuristics to

significantly improve the precision of the abstract interpretation process.

Tool Implementations and Case Studies

Based on the developed semantics these analysis techniques are implemented in two types of tools. *SFCheck* (*S7check*) which support the verification of SFCs as defined in the standard (S7check as defined by Siemens Corp) and *Homer* which uses the proposed analysis techniques for IL.

The developed techniques and tools are successfully tested on a number of case studies provided by academic partners as well as by industry.

1.5.3 Summary

We briefly summarize the main contributions of this thesis:

1. Investigation of semantics as defined in IEC 61131-3 and development of a comprehensive formal semantics for SFCs and IL.
2. Verification of SFCs by model checking. Definition of safe SFCs and development of an automatic analysis.
3. Abstract simulation of IL programs.
4. Combination of abstract interpretation and data flow analysis to verify IL programs.
5. Implementation of analysis techniques and application up to industrial size case studies.

1.6 Thesis Outline

In Chapter 2 we give an overview of related work. On the one hand, we give references to other projects in software verification and their proposed methods in general. On the other hand, we present a thorough overview of existing work applying formal methods to PLC and PLC programming languages. We discuss the different approaches and relate these to the content of this thesis.

Chapter 3 provides the reader with the necessary background to verification in general and software verification in particular. Next to basic mathematical terms and notions, we introduce some major software verification and analysis techniques. We start with model checking and temporal logics, present some modeling frameworks, i.e., various kinds of automata, and point to limits and complexity issues of the model checking approach.

Data flow analysis is introduced subsequently. We present a number of classical examples which illustrate the different problems and analysis

techniques. Based on the examples, a general formal framework is presented as well as a general solver that adepts to the various analysis classes.

Next, we present the concept of abstract interpretation. The basic terms relating the different domains of abstractions are defined. Moreover, we define *safe approximations* of one system to the other and go into detail on how to compute fixed points on an abstract level effectively. Next to the formal basics we explain the application of abstract interpretation to program analysis.

Chapter 4 has as theme programmable logics controllers and their programming language semantics. We give an introduction to PLC hard- and software as well as different possible modeling approaches. We focus on two programming languages: sequential function charts and instruction list. The high level programming language SFCs has many convenient features; however, its semantics is far from obvious. We highlight a number of ambiguous or undefined cases before introducing a formal syntax and semantics for SFCs. The semantics is aimed to be as flexible as possible with respect to different interpretations, while at the same time providing a uniform formal framework. Moreover, an extension from untimed to timed SFCs is sketched.

On the other hand we introduce the IL language. We define the syntax of IL programs for a relevant subset of instructions. Moreover, a structural operational semantics for this subset is defined.

The different verification approaches to the PLC languages defined in the previous chapter are the subject of Chapter 5. We present a method to model check SFCs. This method relies on a translation from SFC source code to the native language of the *Cadence SMV model checker.* We define this translation in detail and illustrate it by an example process taken from chemical engineering. Moreover, we are concerned about structural properties of SFCs and introduce the notion of *safe SFCs.* We slightly modify the previous translation from SFCs to the model checker and generate temporal logic properties to check for safe SFCs.

For IL programs we define an abstract semantics which can cope with sets (intervals) of numbers and three-valued logic. Based on the abstract semantics we introduce the notion of abstract simulation. Moreover, we apply the abstract interpretation approach to program analysis. This, however, is based on a fixed point computation; we point at different ways to compute them efficiently, resulting in the application of *weak topological orders.* Moreover, due to the specific nature of IL programs, the abstract interpretation often results into a gross over-approximation in which standard narrowing techniques do not work. Therefore, we introduce a heuristics based on constraint solving to improve the precision of the analysis results.

Subsequently, we present a number of applications of data flow analysis techniques to program analysis. These are mostly based on the previous abstract interpretation results and we demonstrate how these results can

improve the precision of the data flow analysis significantly.

Chapter 6 presents two tools which are a result of the previous considerations. One tool is called *SFCheck*, which is an implementation of the translation from SFC code to the model checker input language, the checking for safe SFCs, as well as various static properties. The other tool is called *Homer* and is an implementation of verification techniques for IL programs. A number of generic properties are checked. The capabilities of both tools are demonstrated by a number of case studies.

A summary of achievements as well as possible directions for future research spawning from this thesis are presented in Chapter 7. Moreover, the subsequent appendix provides more details about the case studies.

1.7 Bibliographic Notes

All chapters of this thesis are based, at least to some extent, on earlier publications. The model checking introduction of Chapter 3 is based on [HLFE02]. The overview of PLC hardware and software in Chapter 4, as well as the classification of related work in Chapter 2, is based on [MH02]. Moreover, the semantics of SFCs in Chapter 4 are covered by ideas and definitions in [BHLL00a], [BH02] and [BHL02]. The translation of SFCs to the input language of the model checker in Chapter 5 is based on [BH01]. The application of abstract interpretation to IL programs, as well as a technique to enhance precision stems from ideas of [BHLL00b]. The brick sorting case study of Chapter 6 is based on [Huu02].

A general overview of related work is presented in the subsequent Chapter 2. It includes and extensive overview of publications in the area of PLC (software) verification and points also to other software verification projects and publications not in the area of PLCs. More specific references which do not relate to the general theme of this thesis can be found in each chapter.

1.8 Acknowledgments

For his supervision of this thesis I thank Yassine Lakhnech, and Willem-Paul de Roever for cherishing a successful research environment over the last couple of years. Moreover, I thank Oded Maler for being the co-referee of this work and also for many fruitful discussions at Verimag as well as during the VHS[1] project.

Most vital have been the collaborations with my colleague Ben Lukoschus and my direct project partners Nanette Bauer[2] and Goran Frehse[3]. With

[1]European Union Esprit-LTR project 26270 VHS (Verification of Hybrid Systems).

[2]DFG project Integration of Software Specification Techniques in Engineering Applications, LA 1012/6-1.

[3]DFG project Continuous-Dynamic Systems KONDISK, LA 1012/5-1.

them I shared a lot of hard work as well as many enjoyable moments at various occasions. Thank you very much!

My colleagues Martin Steffen, Karsten Stahl, Kai Baukus and Marcel Kyas have been a great help for me, discussing current research as well as helping me to solve various problems over the last years. I appreciate that together with Ben Lukoschus they also took over the painful job of proof reading.

The cooperation with my colleagues at Verimag and with my partners in the various projects I belonged to has been eclectic and stimulating. Among many others, I like to thank in particular Angelika Mader, Gordon Pace, and Sébastien Bornot.

I am grateful to Birgit Vogel-Heuser for providing me with an important case study. Moreover, I thank Wai Wong who enabled me to continue our joint research at Hong Kong Baptist University.

Above all, I wholeheartedly thank Reinhard and Erika, as well as Andrea and Ernst.

Chapter 2

Related Work

This chapter provides an overview over different approaches to algorithmic software verification. Section 2.1 surveys a number of recent contributions software verification in general while different approaches for PLC verification are covered in Section 2.2. In Section 2.3 connects of the surveyed work with this thesis are discussed.

2.1 Software Verification in General

In this section we give reference to a number of projects concerned about algorithmic software verification in general. We cluster the references by projects rather than by methods and techniques to give some ideas about the combinations of different approaches. Moreover, the related projects can be roughly divided in pure abstract interpretation and static analysis approaches and approaches where the main emphasis is on model checking.

2.1.1 Focus on Model Checking

The Bandera [CDH+00] project addresses one of the major obstacles in the path of practical software verification. Tools like model checkers accept as input a description of a finite state transition system, however, models and even more real life systems are often not finite state. The goal of the Bandera project is to integrate existing programming language processing techniques with techniques to provide automated support for the extraction of safe, compact, finite-state models from Java source code that are suitable for verification. Therefore, a number of techniques such as program slicing [DH99], program abstractions [DHJ+01] and other static analysis techniques are combined [CDH01].

The SLAM [BR02] project at Microsoft Research has created an automated process for finding errors in C programs, driven by a user-supplied (temporal safety) property. The process requires that the user states the

property of interest. Then, based on given property, abstractions of the C code are automatically created and analyzed. The process is realized in the SLAM toolkit, which supports the creation, analysis and refinement of abstract models for C code. The toolkit iteratively refines the model until it is precise enough to find a true defect or to validate the given property. The key idea is to use predicate abstractions and abstraction refinements [BMR02].

The FeaVer project [fea] is based on model checking software with the help of the model checking tool SPIN [Hol97]. It has given rise to a stand-alone tool, named FEAVER, for verification of general software written in ANSI-C. The user's responsibility is confined to maintaining a database of properties that the software is required to satisfy, and to interpreting the error scenarios that the system produces. The main components of the Feature Verification System [HS00] are: A library of properties that captures the verifiable properties of the code being checked, a model extraction capability that can derive abstracted verification models mechanically from implementation level C code, and a model checker running in the background to check the source against the properties. In the same context there is a source code analyzer called UNO under development that statically finds the following bugs: use of uninitialized variables, nil-pointer dereferencing, and out-of-bound array indexing [Hol02].

The Java Path Finder (JPF) [HP00] is a tool-set developed at NASA Ames to verify Java code. It is a verification and testing environment for Java which integrates model checking, program analysis and testing. JPF is the second Java Model Checker developed by the Automated Software Engineering group at NASA Ames - JPF1 used a translation from Java to the input language of the model checker SPIN.

The Symbolic Analysis Laboratory (SAL) [BGL+00] is based on a common platform where various components for different tasks can be plugged in. In [PSSD00] initial results in model checking multi-threaded Java programs are presented. Java programs are translated into the SAL intermediate language, which supports dynamic constructs such as object instantiations and thread call stacks. The SAL model checker then exhaustively checks the program description for deadlocks and assertion failures. Basic model checking optimizations that help to curb the state explosion problem are implemented as well.

2.1.2 Focus on Abstract Interpretation

The PolySpace Verifier[1] for C and Ada programs is a commercial tool designed to directly detect run-time errors and non-deterministic constructs in applications written in the respected programming languages at compila-

[1] PolySpace Technologies. `http://www.polyspace.com/`

tion time. The PolySpace Verifier pinpoints the faulty code section that will cause a run-time error if the application was executed. The core technique used in this tool is abstract interpretation [Deu94].

PolySpace Verifier targets at checking automatically and exhaustively for the following errors: Attempt to read a non-initialized variable access conflicts for unprotected shared data in multi threaded applications, referencing through nil-pointers, buffer overflows such as out-of-bounds array access and out-of-bounds pointers, illegal type conversion, invalid arithmetic operations, and unreachable code [Pol02]. However, most algorithms and ideas are not publicly available which makes comparison to other approaches difficult.

In the same context there exists an abstract debugger for Pascal programs [Bou92]. The debugger uses abstract interpretation to check annotations of programs.

The AbsInt[2] company makes use of compiler technology for microcontrollers and digital signal processors. In particular they focus on program optimization and the prediction of worst case execution times [FHL+01]. The analysis requires some program annotations, e.g., the maximum number cycles of a loop.

2.1.3 Other Approaches

A tool that uses neither model checking nor abstract interpretation techniques is the *Compaq Extended Static Checker for Java* [DLNS98]. Java programs are annotated in a given language, i.e., invariants are specified. From these verification conditions are generated and the ESC tool checks for various generic properties such as array bound errors, NIL pointer dereferences, division by zero etc. In contrast to the other discussed approaches a theorem prover is used for the underlying decision process [FLL+02].

Static analysis heuristics are used in the Splint (previously known as LCLint) project [Eva96]. ANSI C programs can be annotated by pre- and postconditions and, based on this, are checked for consistency. Moreover, Splint can detect various program vulnerabilities such as buffer overflows [EL02].

2.2 PLC Verification

This section presents various different ideas and approaches to PLC verification. We give a comprehensive list of different approaches to all PLC programming languages, even though not all of these languages are subject to this thesis. This helps to relate our approach in the context of PLC verification and might give rise to future research projects. A different survey

[2] AbsInt GmbH. http://www.absint.com

can be found in [RLR99].

The order of the listed references is insignificant, however, we cluster the approaches by the PLC languages examined in the related papers. Moreover, we classify the approaches by what is modeled, i.e., the stand alone program or including its environment. Moreover, in with respect the cyclic behavior is treated: In the reviewed work either it is neglected or the execution on the machine (PLC) is modeled as well. This can be done *explicitly* by including real-time for each cycle or instruction, or *implicitly* when the cycle is considered on a more abstract level as a clock tick or alike. A more comprehensive overview of these criteria cam be found in Section 4.2 A matrix directly relating the publications with these criteria is displayed at the end of this section.

2.2.1 Approaches Based on Instruction List

Timed automaton. In [MW99] a timed automaton semantics is given to Instruction List. The language fragment does not contain function and function block calls. Timers can be modeled, but variables are restricted to Booleans. The scan cycle is modeled explicitly, with lower and upper time bound.

Based on this work a tool was developed as described in [Wil99]. It translates IL programs to timed automata. The IL programs considered allow bounded integers as variables, too. The resulting timed automata are represented in the format as used by the model checker UPPAAL [LPY97a]. The environment is also modeled as a timed automaton synchronizing with the automaton modeling the PLC program. For model checking UPPAAL is used.

There are two execution mechanisms treated: explicit cyclical execution with time intervals of variable length and instantaneous execution whenever input signals (or timer output) change.

C/E systems. In [Bau98] condition/event-systems (including real-time) are used to model fragments of IL programs without jumps. Data-types are restricted to Booleans. Timers can be modeled, under the assumption that they are started only at the beginning of a program. The scan cycle is modeled explicitly, but with constant cycle length. The environment is also modeled as a condition/event-system. The verification of the whole system can be done with help of the tool VERDICT [KBP+99]. VERDICT provides an interface to already available verification tools, such as KRONOS [OY93a], HyTech [HHWT97] and SMV [McM00].

In [BKT] this work is extended by including also jumps and a representation of scan cycles with variable cycle length. There is a duration assigned to each instruction and the duration of a whole scan cycle results from the

durations of all the instructions executed in this particular program execution.

Petri nets. In [HM98] IL programs are modeled as Petri nets. The supported language fragment includes the standard set of instructions without commands from libraries. Possible data structures are anything that can be coded within a 8-bit word. Real-time, however, is not represented in the model and the execution mechanism relies on the Petri nets semantics. The properties that can be verified are those expressible in the verification tools PEP [pep] and SMV.

NCES. Also [HTLW97] deals with Instruction List programs. The models are Net Condition/Event systems extended by time. There is an explicit scan cycle considered with a lower and an upper time bound. The available data structure are Booleans only. Instruction executions are modeled by a sequence of transitions. Furthermore, the only instructions mentioned in the paper are LD(N), AND, OR and ST. Timers can be modeled. However, it is not completely clear how the timer model is related to a timer call in an IL program. There is a tool automatically translating IL programs to timed Net Condition/Event systems.

In [RK98] a translation of net C/E systems (NCES) to the input language of the SMV model checker is proposed.

SMV. The verification of instruction list programs is the also the issue of [CCL+00]. The considered language fragment is a small subset consisting of load, store, loop and basic Boolean operations. The supported data type are Booleans, integers are allowed for comparison only. For this language fragment a semantics and a coding in SMV is sketched. Time and timers are not part of the model, but the cycling behavior is implicitly embedded.

2.2.2 Approaches Based on Sequential Function Charts

Timed automaton. In [LPM95] Sequential Function Chart programs are modeled as timed automata. They abstract from explicit scan cycles, meaning that a change in the environment causes an instantaneous reaction of the program. The use of timers is restricted to a special case: with activation of each step an associated timer is started and transitions between steps may depend on these timers. The data types are restricted to Booleans. There exists a compiler from SFC programs to timed automata and the verification is done with KRONOS [OY93b].

Execution model. SFCs are examined from a control theoretic point of view in [HFL01]. This works examines different possible executions mechanisms of SFCs. In particular alternatives for evaluating transition condition,

activating actions and taking steps are discussed. Explicit execution times for cycles or timers are not considered. Moreover, an approach to translate SFCs to LDs is sketched. The resulting LD has the same untimed behavior as the SFC, but the cycling behavior is a different one.

Domain specific language. In [Tou96] fragments of FBs and SFCs are modeled which is based on domain specific language. Time is included in the model. The semantics is general enough to deal with variable scan cycle length, but in this thesis it is generally assumed that scan cycles are of fixed time. The definition of the semantics provides also proof rules as worked out in [AT98]. Composition with an environment is not discussed.

2.2.3 Approaches Based on Ladder Diagrams

Set constraints. In [AFS98] programs of the language Ladder Diagrams are investigated. Variables are of type Boolean. The programs are modeled as set constraints. Only single program executions are considered. The main correctness property that is checked is absence of races. The model does not include real-time and there is no model covering the environment. Instead, input signals are chosen randomly for a finite number of times, which does not model the complete behavior. The program analysis is done by a general constraint resolution engine.

Transition systems. In [Moo94] PLC programs in LD are investigated. The model is a finite transition system. The Ladder Diagrams are straightforwardly translated to Boolean assignments. The assignments define the transitions between states that represent the values of the Boolean variables. Time is only treated implicitly and the environment is not taken into account. Model checking is done with support of the SMV tool.

In [TPP97] two case studies are presented for chemical processes. The programs are modeled as in [Moo94]. Additionally, the environment is modeled as finite state machine as well. The interaction of the environment and controller is alternating: when the controller has reached a stable state (i.e. the control variables do not change any more), the environment may takes a step which may have consequences on the control variables etc.

Translation to SFCs. The goal of [FK92] is to rediscover the program structure hidden in LD programs by translating these to the more intuitive SFCs. Therefore, the authors present a translation from LDs to SFCs (in fact Grafcet) discussing several design issues. This work does not consider timers, but is concerned about scan cycles. However, programs that are executed within one cycle in LD might be executed in several cycles in the resulting SFCs. There is no explicit notion of cycling time here and the interaction of the programs with the environment in not considered as.

SMV. In [RS00] a formal modeling and verification approach for Ladder Diagrams is presented. The considered ladder diagrams comprise Booleans and particular timers. The semantics is given in terms of SMV code and the execution cycle is explicitly modeled, but no timing information is given for the cycle. Verification is directly based on the chosen semantic model, the SMV code.

2.2.4 Miscellaneous Other Approaches

Duration calculus. The goal of [Die97b] is the derivation of correct PLC programs. Specifications are written in a subset of duration calculus. They can be transformed to a graphical representation, that is called *PLC-automata* [Die97a]. From PLC-automata programs in the language Structured Text can be automatically derived.

The semantics of PLC-automata explicitly models scan cycles. The use of timers is restricted to the special case that certain input may be ignored for a specified time. Real-time behavior can be analyzed for programs using Booleans.

In [DFMV98] a timed automaton semantics is given for PLC-automata. Consequently, PLC-automata can be transformed to timed automata and verified with, e.g., KRONOS. Tool support is available through Moby/PLC [Tap98].

HOL. In [KV97] the authors model specifications and implementations in higher order logics. Function blocks are modeled as relations on streams. According to the framework, there are no restrictions on data types. Time is treated implicitly. In the logical framework controller and environment specifications are simply composed by conjunction. Proofs are done with help of a theorem prover, the Isabelle/HOL system [TN02].

Signal. In order to benefit from representations and analysis tools of the synchronous languages community, a translation from a basic fragment of Structured Text to the relational declarative synchronous language SIGNAL is developed in [JFR99]. In particular representations of assignments, conditionals and bounded loops in SIGNAL are discussed. Execution cycles are considered to be implicit, however, no time or timing behavior is taken into account.

2.2.5 Matrix of Surveyed PLC Papers

Table 2.1 provides a brief overview of the surveyed PLC papers according to the criteria discussed. The following abbreviations are used:

Table 2.1: Matrix of surveyed PLC papers

paper	language	domain	cycle	model	method
[MW99, Wil99]	IL	prg.	explicit	TA	MC
[Bau98, BKT]	IL	prg.	explicit	TCES	MC
[HM98]	IL	prg.	implicit	PN	MC
[HTLW97, RK98]	IL	prg. + env.	explicit	NCES	MC
[CCL$^+$00]	IL	prg.	implicit	SMV	MC
[LPM95]	SFC	prg.	implicit	TA	MC
[HFL01]	SFC	prg.	implicit	LD	–
[Tou96, AT98]	SFC, FB	prg.	explicit	DSL	TP
[AFS98]	LD	prg.	implicit	SC	CR
[Moo94]	LD	prg.	implicit	TS	MC
[TPP97]	LD	prg. + env.	implicit	TS	MC
[FK92]	LD	prg.	implicit	SFC	–
[RS00]	LD	prg.	explicit	SMV	MC
[Die97b, Die97a, DFMV98]	PLC	PLC + env.	explcit	DC, TA	MC
[KV97]	FB	prg. + env.	explicit	HOL	TP
[JFR99]	ST	prg.	implicit	SIGNAL	–

Language: FB = function block, LD = ladder diagram, IL = instruction list, PLC = model of the PLC itself, SFC = sequential function charts, ST = structured text.

Domain: prg. = program, env. = environment.

Model: code = program code itself, DC = duration calculus, DSL = domain specific language, HOL = higher order logic, LD = ladder diagram, NCES = net condition/event systems, PN = Petri nets, SC = set constraints, SIGNAL = SIGNAL code, SMV = SMV code, TA = timed automata, TCES = timed condition/event systems, TS = general transition systems.

Verification method: CR = constraint resolution, MC = model checking, TP = theorem proving.

2.3 Discussion

Let us distinguish between the semantics and verification aspects of the PLC languages, namely, SFCs and IL, we consider in this work.

Concerning the semantics of IL there is no other approach that provides an SOS semantics for this language. In fact, most approaches are based on modeling IL programs in terms of (timed) automata, Petri nets, and NCES. These models are mostly abstractions or simplifications of the actual language. This holds even more for SFC semantics. There is no work that gives a comprehensive formal semantics for SFCs including hierarchy, priorities on transitions, actions and action qualifiers. For both languages this thesis provides a significant contribution for the understanding and formal modeling of the considered languages.

There has been little work in providing verification methods for SFCs, yet. The only algorithmic approach in [LPM95] deals with simplified SFCs only, not taking hierarchy, priorities or activity manipulation into account as it is pursued in this work. On the other hand, the use of model checking is common amongst the mentioned algorithmic software verification groups.

Model checking is also used for all verification approaches dealing with IL programs. While this provides more freedom in defining program specific verification properties in contrast to the generic and program independent properties we check in this work, it constraints to (unrealistic) simple IL programs or abstract models only. Potentially infinite data structures such as integers cannot be treated well by model checking without (hand made) abstractions or bounding the range of the integers a priori. Although bounding of ranges leads to a finite model, it is likely to be too huge to check realistic programs. On the other hand model checking is useful in timing analysis of cycles, e.g., worst case execution times.

For the IL languages the verification goals closest to our work are the ones of PolySpace, although they are concerned about checking Ada and C code. However, they also advocate a static analysis and abstract interpretation approach. Unfortunately, their exact methods and techniques are not disclosed, hence, we cannot further compare our approach to theirs, e.g., in terms of the underlying data structures or the combinations of verification techniques.

Similar approaches are also pursued by [Bou92] and AbsInt. The language examined by the latter is also close to IL. Both approaches, however, require an annotation of the respective programs. Although this provides the possibility to check more specific properties, it contravenes our idea to keep the human effort as low as possible.

Overall, this work provides the most comprehensive formal semantics for the PLC languages IL and SFCs at present. It is the only work tackling PLC software verification already at source level and the combination of data flow analysis and abstract interpretation is a novel approach to PLC verification. Moreover, this proposed combined verification method has only partially been explored yet and, thus, provides relevant experiences for the design of new verification approaches.

Chapter 3

Foundations of Software Verification

3.1 Introduction

Software verification and validation of its design process has been an issue since the early days of programming. This starts from the problem of defining software requirements and a model for the software design leading to the verification of its implementation and its integration into larger components.

In the current design process some methods to enhance the quality of software have already made their way into industrial standard practice. These comprise techniques such as listing software requirements and striving to a clean documentation for the design process as well as code review, software simulation, and testing to check for the correctness of the implemented software.

However, techniques in common practice have various drawbacks.

- They often lack a formal basis. E.g., the specification is defined in natural language, which easily leads to misunderstanding and misinterpretation.
- The requirements are not complete, i.e., there are cases which are not taken into account. Hence, parts of the system remain unspecified and, thus, are allowed to behave differently than the designer had in mind. Especially without a formal model it is more likely to forget cases, or cases which are defined in a contradictory way remain undetected.
- The verification of the implementation might be approached in an unorganized way. For instance, testing is done with some arbitrary inputs when it is more efficient to choose data according to the boundary conditions of the model or the implementation, simulation focuses on "non-important" variables, and code review neglects inter-procedural dependencies etc.

- Most informal techniques are not exhaustive, i.e., they do not cover all program executions and thus give way to subtle but often fatal flaws.

Formal methods promise to remedy the above mentioned weaknesses. However, formal methods are not fool-proofed in themselves. Sometimes they require exhaustive knowledge in mathematics, logics and the understanding of the system. Moreover, they do not a priori prevent to forget requirements or even ensure to map the real world appropriate to the model. Especially in software design, one often has a clear idea of what to achieve, but much less of how to specify this formally or even to define what is considered to be legal and what to be harmful. Formal methods provide tools to further investigate into the design and verification process and, thus, allow to enhance the quality of software significantly, but they do not buy any guarantee that what you prove is what you have in mind.

In this chapter we lay the ground for the technical background of software verification. In particular, we introduce the necessary mathematics in Section 3.2. In Section 3.3 the technique of model checking is introduced and different kinds of system models are discussed. Static analysis is in the focus of Section 3.4 and Section 3.5. In particular, we concentrate on data flow analysis and abstract interpretation for program analysis.

3.2 Preliminaries

In order to define the key ideas of software verification techniques used throughout this work – such as model-checking and abstract interpretation – we present some basic notions and notations first. Therefore, we give a brief introduction to lattices and fixed points. A more thorough introduction can be found, e.g., in [DP90].

3.2.1 Lattices

Lattice theory is the study of sets of objects known as lattices. It is an outgrowth of the study of Boolean algebras, and provides a framework for unifying the study of classes and ordered sets in mathematics. The study of lattice theory was given a great boost by a series of papers and a subsequent textbook written by Brinkhoff [Bri67].

In our context we will use lattices to define certain structural properties of programs, e.g., the arrangement of control points within a program, and to reason about computation domains. We start with some basic definitions.

Definition 3.1 (partial order, poset)
*A binary relation $\sqsubseteq$ on a set P is a **partial order** if and only if it satisfies for all $x, y, z \in P$ the following conditions:*

1. $x \sqsubseteq x$ *(reflexivity),*

2. $x \sqsubseteq y$ *and* $y \sqsubseteq x$ *implies* $x = y$ *(anti-symmetry), and*

3. $x \sqsubseteq y$ *and* $y \sqsubseteq z$ *implies* $x \sqsubseteq z$ *(transitivity).*

The ordered pair $\langle P, \sqsubseteq \rangle$ *is called a* poset *(partially ordered set) when* $\sqsubseteq$ *is a partial order on* P*.*

The partial order relation $\sqsubseteq$ can be pronounced "approximates". It imposes some order on the members of P, but is less restrictive than a total order (see below). In particular, it is possible for two members of P to be incomparable, i.e., for neither $x \sqsubseteq y$ nor $y \sqsupseteq x$ to hold, in contrast to:

Definition 3.2 (total order, chain)
A binary relation $\sqsubseteq$ *on a set* P *is a* total order *if and only if it is*

1. *a partial order, and*

2. *for any pair of elements* x *and* y *of* P*,* $(x, y) \in \sqsubseteq$ *or* $(y, x) \in \sqsubseteq$*.*

That is, every element is related with every element one way or the other. A total order is also called a linear order. The ordered pair $\langle P, \sqsubseteq \rangle$ *is called a* chain *or a* totally ordered set *when* $\sqsubseteq$ *is a total order.*

Since an order defines something like "greater-than" or "less-than" we can also think of an maximal or minimal element.

Definition 3.3 (minimal/maximal element)
Let $\langle P, \sqsubseteq \rangle$ *be a poset. An element* $y \in P$ *is a* minimal element *of* P *if there is no element* $x \in P$ *that satisfies* $x \sqsubseteq y$*. Similarly, an element* $y \in P$ *is a* maximal element *of* P *if there is no element* $x \in P$ *that satisfies* $y \sqsubseteq x$*.*

A minimal or maximal does not necessarily exist, consider for instance the set of numbers $\mathbb{Z}$ with the natural order $\leq$. Even if it exists it does not necessarily have to be unique. The unique elements are defined as follows:

Definition 3.4 (least/greatest element)
Let $\langle P, \sqsubseteq \rangle$ *be a poset. Then an element* $y \in P$ *is the* least element *of* P *if for every element* $x \in P$ *we have* $y \sqsubseteq x$*. Similarly an element* $y \in P$ *is the* greatest element *of* P *if for every element* $x \in P$ *we have* $x \sqsubseteq y$*.*

Note that the least/greatest element of a poset is unique if one exists. This is due to the anti-symmetry of $\sqsubseteq$. We call the greatest element *top*, denoted by $\top$, and the least element *bottom*, denoted by $\bot$. The bottom element is often referred to as "undefined". Although the bottom element of every different poset is also different, we use the single symbol $\bot$ to denote all of them, if it is clear from the context. Otherwise, we use $\bot_P$ to denote the bottom element of $\langle P, \sqsubseteq \rangle$.

Many important properties of an ordered set $\langle P, \sqsubseteq \rangle$ can be expressed in terms of the existence of certain upper or lower bounds of subsets of P. Lattices and complete lattices are also characterized in this way. But first let us define the notions of upper and lower bounds.

Definition 3.5 (lower/upper bound)
Let $S \subseteq P$ be a subset of a poset $\langle P, \sqsubseteq \rangle$. An element $p \in P$ is called lower bound *of S if for any element $s \in S$ we have $p \sqsubseteq s$. Conversely, $p \in P$ is called* upper bound *of S if $s \sqsubseteq p$ for any element $s \in S$.*

In contrast to a minimal (maximal) element, the lower (upper) bound of a poset does not necessarily belong to the set itself but to a superset of it. Since the superset can be arbitrarily large, there may be more than one lower (upper) bound. To characterize the lower (upper) bound which is "closest" to the considered poset we define:

Definition 3.6 (least upper bound, greatest lower bound)
Let $S \subseteq P$ be a subset of a poset $\langle P, \sqsubseteq \rangle$. An element $x \in P$ is called least upper bound *of S, denoted by $\mathsf{lub}S$, if*

1. *x is an upper bound of S, and*
2. *$x \sqsubseteq y$ for all upper bounds y of S.*

Similarly, an element $x \in P$ is called greatest lower bound *of S, denoted by $\mathsf{glb}S$, if*

1. *x is an lower bound of S, and*
2. *$y \sqsubseteq x$ for all lower bounds y of S.*

The greatest lower bound of a set S is sometimes also called the *infimum* of S and the least upper bound the *supremum* of S. In general, the least upper bound (greatest lower bound) of a poset does not necessarily exist. It does, however, for chains. For convenience we introduce the following notations:

$$\begin{array}{lll} x \sqcup y & \text{for} & \mathsf{lub}\{x, y\}, \\ x \sqcap y & \text{for} & \mathsf{glb}\{x, y\}, \\ \bigsqcup S & \text{for} & \mathsf{lub}S, \text{ and} \\ \bigsqcap S & \text{for} & \mathsf{glb}S. \end{array}$$

Sometime we call $\sqcap$ the *meet* operation and $\sqcup$ the *join* operation. Based on these notations we define a lattice as follows:

Definition 3.7 (lattice, complete lattice)
Let $\langle P, \sqsubseteq \rangle$ be a non-empty poset. We call $\langle P, \sqsubseteq \rangle$ a lattice, *if $x \sqcup y$ and $x \sqcap y$ exist for all $x, y \in P$. We call $\langle P, \sqsubseteq \rangle$ a* complete lattice, *if $\bigsqcup S$ and $\bigsqcap S$ exist for all subsets $S \subseteq P$.*

A useful property for lattices and posets in general is to know whether they are distributive, i.e., the least upper bound distributes over the greatest lower bound relation.

Definition 3.8 (distributive)
A complete lattice $\langle L, \sqsubseteq \rangle$ *is* distributive *if for all* $x, y, z \in L$ *we have* $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$.

Another interesting property is the existence or, more precise, non-existence of infinite strict ascending chains.

Definition 3.9 (ascending chain condition)
Let $\langle P, \sqsubseteq \rangle$ *be an ordered set.* P *satisfies the* ascending chain condition *(ACC), if for any chain* $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ *in* P *there exists* $k \in \mathbb{N}$ *such that* $x_k = x_{k+1} = \cdots$.

This means, any ascending chain which satisfies ACC has a greatest upper bound and it is an element of the chain.

In order to illustrate some of the introduced terms consider the following example.

Example 3.1 Consider the set of natural numbers with the standard "less-than" order $\langle \mathbb{N}, \leq \rangle$. Obviously, "less-than" is a total order and, hence, $\langle \mathbb{N}, \leq \rangle$ is a chain. Moreover, $\mathbb{N}$ has a minimal element, namely zero, which is at the same time the least element. But $\mathbb{N}$ does not have any maximal or greatest element since there is not even an upper bound in $\mathbb{N}$.

However, we can augment $\mathbb{N}$ by a distinct element $+\infty$ such that $\mathbb{N}' = \mathbb{N} \cup \{+\infty\}$ and extend the order $\leq$ to $\leq'$ such that for any $n \in \mathbb{N}'$ we have $n \leq +\infty$. Then, the resulting lattice $\langle \mathbb{N}', \leq' \rangle$ does have a maximal and greatest element (which is of course $+\infty$). Moreover, $\langle \mathbb{N}', \leq' \rangle$ is a complete lattice, while $\langle \mathbb{N}, \leq \rangle$ is a lattice, but not a complete one. $\langle \mathbb{N}, \leq \rangle$ does not form a complete lattice since it does not always satisfy the ACC, e.g., the subset of all even numbers does not have a greatest element in itself or even in $\mathbb{N}$.

For the remainder of this work we mostly consider complete lattices only.

3.2.2 Complete Partial Orders

Complete partial orders are often required as a minimum algebraic structure in order to ensure some well-behavior or the solution to certain equations as shown in Section 3.2.3.

Definition 3.10 (directed, consistent)
Let S *be a non-empty subset of an ordered set* $\langle P, \sqsubseteq \rangle$.

- S *is* directed*, if for every finite subset* F *of* S *there exists an upper bound* z *of* F *such that* $z \in S$.
- S *is* consistent*, if for every finite subset* F *of* S *there exists an upper bound* z *of* F *such that* $z \in P$.

It is easy to see that non-consistency arises only in ordered sets without a top element. Moreover, every directed set is consistent.

Definition 3.11 (CPO)
An ordered set $\langle P, \sqsubseteq \rangle$ *is a* complete partial order (CPO) *if*

1. *P has a bottom element, and*
2. *for every directed subset D of P the least upper bound $\bigsqcup D$ exists.*

Note that every complete lattice is a CPO. The converse does not hold since it does not require any greatest lower bound.

3.2.3 Fixed Points

A point that is mapped to itself within an order-preserving selfmap is generally called a *fixed point.* Fixed points are a long studied subject and there has been a classical question concerning fixed points:

> Characterize those (finite) ordered sets that have the fixed point property.

The recorded history of this problem seems to start in the papers by Knaster in the twenties and then by Tarski and by Davis in the fifties, where the questions are answered for lattices. However, the vague formulation of the problem has inspired many possible approaches [Sch99].

We present the basic background and the most prominent results here. Therefore, we first define some properties on mappings that help to reason about fixed points and their existence in the latter.

Definition 3.12 (monotone, continuous, strict)
Let $\langle P, \sqsubseteq_P \rangle$ *and* $\langle Q, \sqsubseteq_Q \rangle$ *be ordered sets. Let* $\phi : P \to Q$ *be a mapping. This mapping* ϕ *is called*

- monotone, *if* $x \sqsubseteq_P y$ *implies* $\phi(x) \sqsubseteq_Q \phi(y)$,
- continuous, *if for every directed set* $D \subseteq P$ *we have* $\phi(\bigsqcup D) = \bigsqcup \phi(D)$, *where* $\bigsqcup \phi(D) := \bigsqcup \{\phi(x) \mid x \in D\}$,
- strict, *if* $\phi(\bot) = \bot$.

A monotone mapping is sometimes called order preserving or isotone. Note that if P has no comparable members, any ϕ is trivially monotone.

Since P is complete, we know that for every directed subset D of P the least upper bound $\bigsqcup D$ exists. This definition is saying, as a side effect, that $\bigsqcup \phi(D)$ also exists for a continuous function. Moreover, a continuous function is limit-preserving. The limit of a continuous function evaluated on a chain is equal to the function evaluated at the limit of the chain. The following lemma shows that indeed any continuous function is monotone.

Lemma 3.1
Every continuous mapping is monotone.

Proof Let $\phi : P \to Q$ be continuous, and let $x \sqsubseteq y \in P$. Since $x \sqcup y = y$, and since ϕ is continuous, $\phi(x) \sqsubseteq (\phi(x) \sqcup \phi(y)) = \phi(x \sqsubseteq y) = \phi(y)$. This means $x \sqcup y$ implies $\phi(x) \sqsubseteq \phi(y)$, hence, ϕ is monotone. ∎

Based on the previous definitions and results we define fixed points and present a number of known results.

Definition 3.13 (fixed point)
Let $\langle P, \sqsubseteq \rangle$ be an ordered set and $\phi : P \to P$ a mapping. A fixed point equation *is an equation of the form*

$$\phi(x) = x$$

in which $x \in P$ satisfying the equation is called a fixed point *of ϕ. The set of all fixed points is*

$$\mathsf{fix}(\phi) := \{x \in P \mid \phi(x) = x\}.$$

The least fixed point *of ϕ is defined by*

$$\mu(\phi) := \min \mathsf{fix}(\phi)$$

if it exists. Similarly, the greatest fixed point *of ϕ is defined by*

$$\nu(\phi) := \max \mathsf{fix}(\phi)$$

if it exists.

Subsequently, we give some theorems on the existence of fixed points.

Theorem 3.1 (Knaster-Tarski)
Let $\langle L, \sqsubseteq \rangle$ be a complete lattice and $\phi : L \to L$ a monotone mapping. Then, $\bigsqcup\{x \in L \mid x \sqsubseteq \phi(x)\} \in \mathsf{fix}(\phi)$.

Proof Let $G = \{x \in L \mid x \sqsubseteq \phi(x)\}$ and let $g = \bigsqcup G$ which exists by the definition of complete lattice. Moreover, by definition $x \sqsubseteq g$, for all $x \in G$. Since ϕ is monotone $x \sqsubseteq \phi(x) \sqsubseteq \phi(g)$. Thus $x \sqsubseteq \phi(g)$ for all $x \in G$ and, therefore, $\phi(g)$ is an upper bound of G. Since g is the least upper bound of G we have in particular $g \sqsubseteq \phi(g)$.

Since ϕ is monotone, $\phi(g) \sqsubseteq \phi(\phi(g))$. Hence, $\phi(g) \in G$ and, thus, $\phi(g) \sqsubseteq g$ since g is the least upper bound of G. Together with the fact that $g \sqsubseteq \phi(g)$ yields $g = \phi(g)$, i.e., $g \in \mathsf{fix}(\phi)$. ∎

Theorem 3.2
Let $\langle L, \sqsubseteq \rangle$ be a complete lattice and $\phi : L \to L$ a monotone mapping. Then, ϕ has a least fixed point $\mu(\phi)$ and $\mu(\phi) = \bigsqcap\{x \in L \mid \phi(x) = x\} = \bigsqcap\{x \in L \mid \phi(x) \sqsubseteq x\}$.

Proof Let $G = \{x \in L \mid \phi(x) \sqsubseteq x\}$ and let $g = \bigsqcap G$ which exists by the definition of complete lattice. Similarly, let $G' = \{x \in L \mid \phi(x) = x\}$ and let $g' = \bigsqcap G'$. In order to show that $g = g'$ is the least fixed point of ϕ we show that $g \in G$,$g \in \mathsf{fix}(\phi)$, and $g = g'$.

1. By the definition of greatest lower bound $g \sqsubseteq x$, for all $x \in G$. Since ϕ is monotone and $x \in G$ we have $\phi(g) \sqsubseteq \phi(x) \sqsubseteq x$, for all $x \in G$. Hence, $\phi(g) \sqsubseteq x$, for all $x \in G$, and by the definition of greatest lower bound, $\phi(g) \sqsubseteq g$ which means that $g \in G$.

2. From the above we know that $\phi(g) \sqsubseteq g$, hence, it remains to show $g \sqsubseteq \phi(g)$ to prove that $g \in \mathsf{fix}(\phi)$. From $\phi(g) \sqsubseteq g$ and the monotonicity of ϕ it follows that $\phi(\phi(g)) \sqsubseteq \phi(g)$. This implies $\phi(g) \in G$. Hence, $g \sqsubseteq \phi(g)$ which yields that $g \in \mathsf{fix}(\phi)$.

3. Since $g \in \mathsf{fix}(\phi)$ and $g' = \bigsqcap G'$ we know that $g' \sqsubseteq g$. On the other hand, we have $G' \subseteq G$, thus, $g \sqsubseteq g'$. This yields $g = g'$ and completes the proof.

■

Theorem 3.3
Let $\langle P, \sqsubseteq \rangle$ be a CPO and $\phi : P \to P$ a mapping such that $x \sqsubseteq \phi(x)$ for all $x \in P$. Then, ϕ has a fixed point.

Theorem 3.4
If $\langle P, \sqsubseteq \rangle$ is a CPO and $\phi : P \to P$ a monotone mapping then ϕ has a least fixed point.

The proofs for the last two theorems are far from trivial and can be found in [DP90]. Note also that the proof of Theorem 3.4 requires the Axiom of Choice.

While the previous theorems stated the existence of fixed points under certain conditions, the following gives also a constructive way to fixed points:

Theorem 3.5 (Kleene)
Let $\langle P, \sqsubseteq \rangle$ be a CPO, let $\phi : P \to P$ be a monotone mapping, and define $\alpha := \bigsqcup_{n \geq 0} \phi^n(\bot)$. Then the following two properties hold:

1. *If $\alpha \in \mathsf{fix}(\phi)$ then $\alpha = \mu(\phi)$.*

2. *If ϕ is continuous then $\alpha = \mu(\phi)$.*

Proof

1. Certainly, $\bot \sqsubseteq \phi(\bot)$, and since ϕ is monotone, we have $\phi^n(\bot) \sqsubseteq \phi^{n+1}(\bot)$. Hence there is a chain

$$\bot \sqsubseteq \phi(\bot) \sqsubseteq \cdots \sqsubseteq \phi^n(\bot) \sqsubseteq \phi^{n+1}(\bot) \sqsubseteq \cdots$$

in P. Since P is a CPO, $\alpha := \bigsqcup_{n\geq 0} \phi^n(\bot)$ exists. Let $\beta \in \mathsf{fix}(\phi)$. By induction, $\phi^n(\beta) = \beta$, for all $n \in \mathbb{N}$. By multiple application of monotonicity of ϕ and since β is a fixed point, we have for all n: $\phi^n(\bot) \sqsubseteq \phi^n(\beta) = \beta$. Hence, β is an upper bound of the chain. Consequently, $\alpha \sqsubseteq \beta$ and if $\alpha \in \mathsf{fix}(\phi)$, then $\alpha = \mu(\phi)$.

2. It suffices to show that $\alpha \in \mathsf{fix}(\phi)$. Let ϕ be continuous. Then:

$$\phi(\alpha) = \phi(\bigsqcup_{n\geq 0} \phi^n(\bot)) = \bigsqcup_{n\geq 0} \phi(\phi^n(\bot)).$$

Since $\bot \sqsubseteq \phi^n(\bot)$, for all $n \in \mathbb{N}$, this implies:

$$\bigsqcup_{n\geq 0} \phi(\phi^n(\bot)) = \bigsqcup_{n\geq 1} \phi^n(\bot) = \bigsqcup_{n\geq 0} \phi^n(\bot) = \alpha.$$

■

3.3 Model Checking

Common to every verification task is to prove that a system, a program, or simply an abstract model of a problem satisfies certain requirements. Formally, this is denoted by

$$M \models \varphi,$$

where M is a model of the system, φ is the requirement, and $\models$ denotes the satisfaction relation. Model checking [QS82, CE82] is an algorithmic way to decide whether M satisfies φ. Although any verification approach is based on this, the actual logic or – more general – the formalisms to denote these three items vary a lot. In the following we present some formal models for each of them. We mainly focus on the ones we will use throughout this work.

3.3.1 System Model

In this work we concentrate on verification of reactive systems. These systems communicate with their environment and may often – like operating systems – not terminate. Hence, a model which captures their infinite behavior in a concise way is desirable. Simply specifying their input/output behavior is not sufficient, it is rather interesting to know the internal *state* of a system, too.

Therefore, we start by describing the behavior of a system with some state-based formalism. Such formalisms include Petri nets [Rei85], CSP [BHR84, Hoa85], CCS [Mil80, Mil89], different forms of automata, LOTOS [BB87], SDL [SDL92], etc. In these formalisms, the behavior of the system is composed of a set of components, each described in terms of local

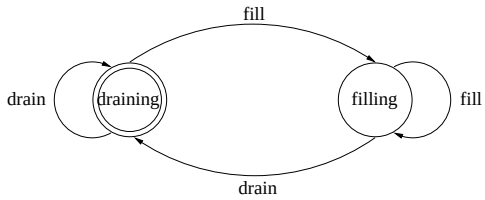

Figure 3.1: Discrete automaton for a tank model

state changes or events. The global behavior of the system is given as the reachable state-space generated from the system description.

In this work we use mainly discrete systems which can be best understood as discrete automata. However, to illustrate their connection to timed and hybrid systems we add a paragraph to each of them.

Discrete Automaton

A discrete automaton $A = (Q, q_0, \delta, F)$ over an alphabet Σ (events, actions) is a structure where

- Q is a finite set of control locations,
- q_0 is an initial location,
- $\delta : Q \times \Sigma \longrightarrow Q$ is a transition function, and
- F is an acceptance condition.

A sequence of actions in Σ which is produced by taking a path through the automaton, starting with the initial location and satisfying the acceptance condition, is called a *word.* The set of all words, i.e., the set of all possible sequences, for an automaton A is called the *language* of A denoted by $L(A)$. The acceptance condition can vary from a single location which indicates the end of the sequence once it is reached to a set of locations which have to be reached infinitely often. The acceptance condition mainly determines the different kinds of discrete automata which can be found in the literature (e.g., [Tho90]).

An example for a discrete automaton is depicted in Figure 3.1. This automaton gives a rough model for a tank. There are two control locations *draining* and *filling*. The double framed circle around *draining* indicates the initial location. Depending on the actions *drain* and *fill* transitions depicted by arrows are taken. We do not give an explicit acceptance condition here and note that the model is very simplified, i.e., it is not captured that the tank might run empty or might overflow.

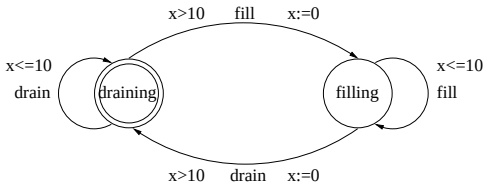

Figure 3.2: Timed automaton for a tank model

Timed Automaton

In contrast to discrete automata, the setting of timed automata [AD94] is in a dense real-time world. To express quantitative time, *clocks* are introduced which are real-valued variables evolving over time. Moreover, they can be checked against thresholds, and they can be reset when a transition is taken.

Formally, a timed automaton over an alphabet Σ is a quadruple $T = (Q, q_0, C, E)$ where

- Q is a finite set of locations,
- q_0 is the initial location,
- C is a finite set of clocks, and
- E is a set of edges of the form (q, γ, a, ρ, q'), where $q, q' \in Q$ are the source and target locations, γ is a *transition condition*, i.e., a Boolean formula over clock variables and thresholds, $a \in \Sigma$ is an action and ρ is the set of clocks that are reset when this transition is taken.

The language of a timed automata is given by the set of all execution sequences over time. Traditionally, infinite sequences are considered where time grows infinitely.

A timed automaton example is depicted in Figure 3.2. In contrast to Figure 3.1 there is a clock x which constraints the moments transitions are taken. This clock serves as a timer for draining and filling periods. Starting from the initial location *draining* the location is changed only if x is greater than 10. If so the x is reset and control will reside in location *filling* until the clock value exceeds 10 again. In the meanwhile self-loops are possible.

Hybrid Automaton

Discrete automata do not incorporate quantitative time, but timed automata do so by the use of clocks. However, they do not model arbitrary continuous functions. This feature is covered by so-called hybrid automata [ACHH93]. These allow do model and reason about a set of continuous variables evolving over time.

Formally, a hybrid automaton $H = (Q, q_0, \mathit{Var}, E, \mathit{Act}, \mathit{Inv})$ over an alphabet Σ consists of

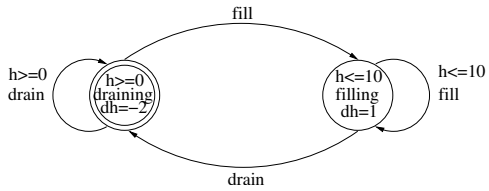

Figure 3.3: Hybrid automaton for a tank model

- a finite set of locations Q with some initial location q_0,
- a finite set of real-valued variables Var,
- a finite set E of discrete transitions, where each transition $e = (q, \rho, a, q')$ between two locations $q, q' \in Q$ is labeled by some action $a \in \Sigma$ and depends on a transition condition ρ which reasons about the variables in Var,
- a labeling function Act that assigns a set of activities to each location $q \in Q$. The activities describe how the variables in Var evolve continuously as long as control resides in q, and
- an invariant $Inv(q)$ for each location $q \in Q$ which defines the terms on the variables in Var for control to reside in q.

The semantics of an hybrid automaton is defined by all trajectories of the continuous variables as well as the actions over time.

A hybrid model for the tank example is shown in Figure 3.3. This model describes the tank level h in a filling and draining process. Draining is two times faster than filling. Although possible, there are no guards or resets on the transitions, but invariants in the locations determine when exactly control is allowed to stay there. Note that there also exist different timed automaton models with invariants, deadlines, and urgency. Mostly this has little effect on the expressiveness (cf. [Sta98]), but allows more or less convenient notations.

3.3.2 Computational Model

In the previous section we described very briefly how to derive a behavior from each description model. However, the models themselves are merely syntax and in order to formally derive a *semantics*, i.e., the system's behavior, the system description can be mapped to a mathematical abstract representation. This abstract representation is also called *computational model* and represents the semantics.

One way to describe a computational model is a state transition system. It consists of states and and has also the ability to represent the fact that in any given state the system reacts to certain actions and might enter new

system states. This pair of system states is then called a *transition*. The semantics of a system is then determined by the sequences of all transitions in a system that start from some given initial state. One formal way to describe these state transition systems are *Kripke structures* [Kri63], named after the logician Saul A. Kripke who used transition systems to define the semantics of modal logics. Transition systems are graphs consisting of states, transitions and a function that maps each state to a set of properties which hold in that state.

Formally, we define a Kripke structure as follows: Given a set of atomic properties P, also called propositions, a Kripke structure $K = (S, S_0, R, \mu)$ contains the following components:

- S is a set of states,
- $S_0 \subseteq S$ is a set of initial states,
- $R \subseteq S \times S$ is a transition relation, which is required to be total, i.e., for every state $s \in S$ there exists an $s' \in S$ such that $(s, s') \in S$, and
- $\mu : S \longrightarrow 2^P$ is a labeling function that assigns a set of propositions to every state.

An execution sequence of a Kripke structure is defined as a possibly infinite sequence $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 \in S_0$ and for every index $i > 0$ in π we have $(s_{i-1}, s_i) \in R$. This means starting from the initial state we go along a path in the graph represented by the Kripke structure. The semantics of a system described by a Kripke structure is the set of all its sequences, i.e., all possible paths from all initial states.

In order to describe the semantics of a system model it is translated into such a computational model first. This means, the system model represents the syntax and the computational model the semantics. For the different types of automata presented above, the computational models are also different. While discrete automata only have to reflect the control location in a state, timed and in particular hybrid systems need to reflect time as well in a state. Since time is dense for both the latter models, it is not always guaranteed to find a finite representation of these systems. However, using abstract or symbolic state representations, i.e., the clustering of concrete states into equivalence classes, in many cases a finite representation is possible also for timed and so called *linear hybrid automata*. The latter are hybrid automata which only allow fixed (but arbitrary) rates for the continuous variables. A finite representation is important in order to guarantee termination for algorithmic approaches like model checking.

3.3.3 Temporal Logics

Describing the system formally is only one thing. For verification it is also necessary to describe the requirements posed to a system in a formal style.

There are different ways to do so. One fundamental issue is to choose between an operational or a declarative way. In this context operational means, e.g., using automata itself in order to specify the desired properties. The advantage is that the same framework for system modeling is also used to specify the system requirements. However, it is often a bit tedious to formulate requirements as automata, and automata are sometimes not as easy to understand as requirements. The declarative way means using logics to specify the requirements.

As mentioned before, we are mainly interested in reactive systems and, therefore, are concerned about the states of a system as well as the transitions between these states. Since basic propositional logic allows to reason about states only but not sequences of states or transitions, so-called *temporal logic* [Pnu81] is used in order to remedy this fact. Temporal logic extends propositional logic, i.e., Boolean proposition with connectivities such as logical conjunction, disjunction and negation, with *modal operators*. These are operators like *always* or *eventually* that allow reasoning over execution sequences and can be combined with the usual connectivities.

Let us define propositional logic first. Based on propositions p logical expressions can be constructed by the following rules:

$$\varphi := p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2$$

Other Boolean connectives like "$\vee$", "$\Rightarrow$", and "$\Leftrightarrow$" can be derived from "$\neg$" and "$\wedge$" as usual.

The semantics is straightforward and we is not shown here. Next, we present the extension from propositional to temporal logic. In general we can define and distinguish between two main temporal sub-logics, namely, linear time and branching time.

Linear Time Temporal Logic

One way to describe requirements is to define desired sequences in time. Linear Time Temporal Logic (LTL) allows to reason about paths in computational models like Kripke structures. In order to do so, propositional logic is extended by the following basic modal operators:

- $\bigcirc$. This denotes the modality "next" and requires that a property holds in the next state of a path, e.g., a path π in a Kripke structure satisfies $\bigcirc\varphi$ if and only if φ is satisfied in the second state of π.

- $\mathcal{U}$. This denotes the infix modality "until". I.e., a path π in a Kripke structure satisfies the expression $\varphi\mathcal{U}\psi$ if and only if ψ is satisfied in some later state of π, and φ holds in all states in between, including the first state of π. This is meant by the expression "φ until ψ".

LTL is founded on these basic modalities and their free combination with propositional logic. From these the following useful abbreviations can be defined:

- $\Diamond$ means "eventually", and a path π satisfies the expression $\Diamond\varphi$ if and only if there exists a state in π which satisfies φ.
- $\Box$ means "always", and a path π satisfies $\Box\varphi$ if and only if all states in π satisfy φ.

Branching Time Temporal Logic

In contrast to LTL branching time logics do not reason over single paths but over sets of paths, more precise, trees. One logic which does so is called Computational Tree Logic (CTL) which is propositional logic extended by *path quantifiers* and *temporal operators*. The temporal operators are the same as in LTL presented above. The path quantifiers are "$\exists$" which requires a single path to exist that satisfies some property and "$\forall$" which requires all paths of the computational model to satisfy some property.

CTL formulas are constructed from propositional logic, temporal operators and path quantifiers in the following way: Every formula starts with a path quantifier, every path quantifier is immediately followed by a temporal operator, and every temporal operator is preceded by a path quantifier.

This allows to build formulas such as

- $\exists\Box\varphi$, which means that there exists a path where always, i.e, for all states, φ holds, and
- $\exists\Diamond\forall\Box\varphi$, which means there exist a path with a certain state from whereon for all paths, i.e., branches, φ is always true.

Remarks

Note that CTL and LTL not only use different means to describe system properties, but in general there are LTL formulas which cannot represented in the CTL framework, and vice versa. Moreover, while linear time appears to be conceptually simpler than branching time, the latter is often computationally more efficient.

For both types of logics there exist real-time extensions. This means the logics provide the possibility to reason about explicit time and distances. We do not go into detail here.

3.3.4 Tools and Limitations

Returning to the initial task of checking $M \models \varphi$, model checking is, as mentioned, an algorithmic (i.e., automatic) way to decide whether a model

M satisfies φ or not. There are several tools supporting model checking. For discrete automata and logics like CTL or LTL there for instance SPIN [Hol97] and SMV [McM00] as well as its improved version CaSMV [JM01] from Cadence Labs we use later in this work. For checking timed automata with real-timed logics there are UPPAAL [LPY97a], KRONOS [OY93a] and extensions of SPIN. For linear hybrid systems HyTech [HHWT97] is a tool that enables to check reachability of certain states of the corresponding linear hybrid automaton. Moreover, there are many more tools which are also based on other system description models as well as logics.

For checking reactive systems one of the presented system models and logics is often used. However, due to fundamental limitations not every model and every logics is applicable for model checking. Timed and even more hybrid systems are restricted to certain classes, since a finite state representation in whatever way has to be guaranteed in order to keep model checking possible, i.e., decidable.

Despite of these basic fundamental restrictions model checking has also to cope with serious complexity issues which are described in the next section.

3.3.5 Complexity Issues

One of the main drawbacks of state-based formal verification methods is the so-called *state explosion problem:* When a large system consists of several smaller components (e.g., automata) running in parallel, the number of global states increases exponentially with the number of components. For instance, consider a system of 20 automata working in parallel, each of which having 10 local states. This amounts to 10^{20} global states. The simple task of enumerating these states on a machine that needs only one nanosecond per state (which is considerably fast at the time of writing) already takes over 3000 years. Building and searching a graph based on these states takes significantly longer and is far beyond today's memory capabilities.

The state explosion problem is inherent in any system having parallel structures and poses a major complexity problem to any verification method based on the exhaustive enumeration of global states. Several techniques have been developed to minimize the impact of this problem on the time and memory consumption of the model checking process. Often a model checking algorithm uses a combination of several such techniques, which are discussed in the following.

Note that although all these methods can result in a significant speedup in practice, they are limited by the worst case complexity inherent to the problem. E.g., model checking LTL or CTL properties for Kripke structures is polynomial in space, in fact $n \log m$ where n is the length of the formula and m is the size of the Kripke structure. When it comes to concurrent programs, i.e., different automata composed in parallel the problem is already

PSPACE-complete even for a fixed formula [LP85, VW94, KVW00]. For an overview of different complexity issues in model checking LTL and CTL formulas we refer the reader to [Var01].

Global vs. Local Strategy

In accordance with the two parameters of the model checking problem, the model M and the requirement φ, there are two basic strategies when designing a model checking algorithm, the "global" and the "local" strategy [Mer01]. "Global" means the algorithm operates recursively on the structure of φ and evaluates each sub-formula over the whole M, while the local strategy checks only parts of the state space at a time but for all sub-formulas of φ. The worst-case complexity of both approaches is the same, however, the average behavior can differ significantly in practice. Traditionally, LTL model checking is based on local approaches while for CTL global algorithms are applied.

"On-the-fly" Techniques

The classical model checking approach builds a complete state transition graph of the system and performs a search on this graph. But often a large part of the graph is not traversed during the search or is even unreachable from the initial state(s) of the search. Therefore it is often a good idea to construct the graph in an "on-the-fly" fashion [CVWY92, BCG95]. That is, only the part of the graph that is currently needed is constructed during the search and kept in memory for later reuse, often supported by caching algorithms.

Efficient Data Structures

A considerable amount of memory can be saved using efficient data structures during the model checking process. One prominent example are *binary decision diagrams* (BDDs) [Bry86, Bry92], which are used as a compact representation of Boolean functions. Ken McMillan suggested in his PhD thesis [McM92] to use them for model checking, and today BDDs and similar data structures are the key solution for efficient memory usage in many kinds of computation software.

In the field of timed automata the observation that despite their continuous nature, clocks are often compared only to each other and a finite and bounded number of constants, opened the possibility to discretize the state space for model checking. So called *clock regions* are stored in data structures like *difference bounded matrices* (DBMs) [Bel57, Dil90] and are used in most model checking tools for timed automata like KRONOS [OY93b] and Uppaal [LPY97b].

Abstraction

Abstraction is a fundamental concept used in all formal verification methods. Abstracting means replacing a concrete object with an abstract one which is more universal, and therefore, often has a simpler structure than before. A well-chosen abstraction simplifies as much as possible, without losing too much information about the concrete object. Abstractions can be used in different ways during the specification and verification process:

- Building the system model: Every translation from a real-life system or an informal system description into a formal model is an abstraction.
- Optimizing the system model: Depending on the property that is to be checked, different abstractions of the system model can be useful, e.g., by abstracting from data, time, or continuous variables to obtain simpler models.
- Reducing the complexity of model checking: Model-checkers often use abstractions to minimize time and space usage, e.g., by introducing symbolic states.

When abstracting a system model, often a so-called *safe abstraction* is chosen: Whenever a property holds for the abstract system, it also holds for the concrete system. The converse, however, does not always hold, due to the over-approximation which occurs in the abstraction process. A positive model checking result on a safe abstraction therefore means that the concrete system also fulfills the property, whereas a negative result can either mean that the concrete system is not correct or that the abstraction is too coarse.

Thus, when getting a negative result, the counterexample provided by the model-checker is examined to see if the error will also occur in the concrete system. If it doesn't, a finer abstraction has to be chosen.

Compositionality

Another important concept is compositionality. In a compositional approach the system model is split into components. Each component is then specified as a single entity, and its correct behavior can be proved by model checking. The specifications of all components are then combined to get the global property of the system model. A prerequisite for this approach is that the behavior of the components is completely described by its specifications such that the behavior of the global system model only depends on these specifications and not on any additional information about the internal structure of the components.

The advantage of such an approach is obvious. Consider the example at the beginning of this section (20 automata, 10 local states each). A compositional approach yields 20 applications of a model checking algorithm,

each of which involving only 10 states, whereas the global approach applies model checking once, but on a set of 10^{20} states. There is, however, some (often significant) overhead for the decomposition of the system model and the construction and the composition of the local specifications.

3.4 Data Flow Analysis

In this section we present a well-known static analysis technique called *data flow analysis* (cf. [NNH99], [Muc97], [ASU86]). It is a flow sensitive method to derive information related to the flow of data along control paths, more precise, for every program point information that summarizes some property of all the possible dynamic instances of that point are computed. It does distinguish between when and how a particular instance is reached.

Data flow analysis originates from compiler construction and was born out of the urge to develop efficient and compact code. For instance, a program containing assignments to a variable which is not used in the latter, i.e., containing *dead code*, is sub-optimal, this code fragment can be eliminated and the resulting program code is more compact.

After defining some basic terms and providing the necessary background for data flow analysis in Section 3.4.1 we present a number of classical data flow problems in Section 3.4.2. Subsequently, in Section 3.4.3 we provide the formal framework for data flow analysis and present two general iterative algorithms for the analysis in Section 3.4.4.

3.4.1 Basic Definitions

As mentioned, data flow analysis is concerned about the flow of data along the control paths of a program. In order to reason about this, we define the basic terms, i.e., we define what is a program, what is a control path and what is a data flow.

A Simple WHILE Language

The kind of programs we consider throughout this section are simple Pascal-like WHILE programs allowing assignments to variables, the empty assignment `SKIP`, while-do-od constructs, and if-then-else conditionals. The semantics should be clear from the context and we do not go into further detail here, since we only use this language for high level demonstration purposes.

A program example is shown in Figure 3.4. There is no particular functionality we are interested in, but we use this program throughout the section for mainly syntactic illustration purposes. It works as follows: The variables `x` and `y` are initialized with the value of `z+1`. Afterwards, while `x` is less than 10, `x` is doubled in every iteration and depending on whether `x` equals the

square of y, y is set to either z+z or z*z. If control leaves the while-loop, the empty statement SKIP is executed. Again, we are not so much interested in the semantics of this program, but the flow of data as illustrated in the next sections.

```
x:=z+1
y:=z+1
WHILE x<10 DO
  x:=x+x
  IF x=y*y
    THEN y:=z+z
    ELSE y:=z*z
OD
SKIP
```

Figure 3.4: Program example

For convenience we define for any program P the following abbreviations:

- Var_P as the set of variables in P,
- Expr_P as the set of all arithmetic expressions occurring in P, and
- Assign_P as the set of assignments occurring in P.

If the program we are referring to is clear from the context we may freely omit the program index. Moreover, if not stated otherwise we use the term *expression* as an abbreviation for *arithmetic expression.*

Flow Graphs

In general data flow problems can be interpreted as information flow problems on graphs. The vertices are *elementary blocks* and the edges describe how control might pass from one block to the other.

Definition 3.14 (elementary block)
Any single assignment or Boolean test is called an elementary block.

We assume there is a unique identifier associated to each elementary block. Sometimes we want to reduce the number of vertices in the graph by not reasoning about elementary blocks but *basic blocks.*

Definition 3.15 (basic block)
A basic block *is a maximal sequence of assignments which are executed sequentially. In particular it does not contain loops or branching. The set of all identifiers of elementary and basic blocks of a program P is denoted by* Block_P.

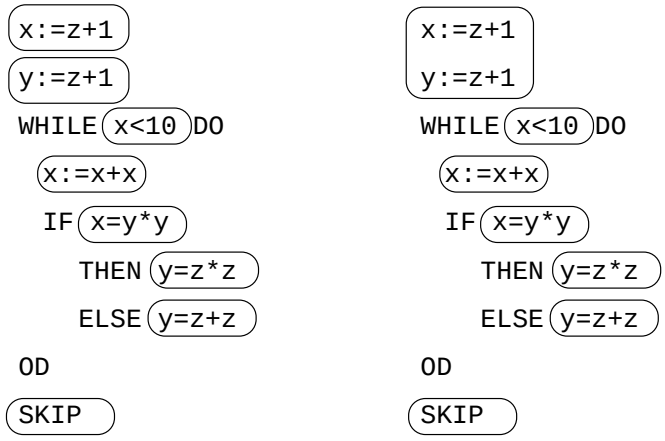

Figure 3.5: Partitioning in elementary and in basic blocks

For the example program of Figure 3.4 a partitioning into elementary blocks and into basic blocks is depicted in Figure 3.5. Sometimes we do not represent a partitioning graphically but print the identifiers of blocks as labels to their statements. For the program in Figure 3.4 this is depicted in Figure 3.6. In the latter when referring to a certain label we in fact refer to the according statement.

```
[x:=z+1]^1
[y:=z+1]^2
WHILE [x<10]^3 DO
  [x:=x+x]^4
  IF [x=y*y]^5
    THEN [y:=z+z]^6
    ELSE [y:=z*z]^7
OD
[SKIP]^8
```

Figure 3.6: Program example with labeled blocks

The vertices of a flow graph are blocks while the edges is a relation on blocks representing the flow of control.

Definition 3.16 (flow graph, flow, reverse flow)
A flow graph $G = (N, E, n_{ini}, n_{fin})$ *is a graph built from of a set of vertices* N *consisting of elementary or basic blocks, a set of edges* $E \subseteq N \times N$*, a distinct initial vertex* $n_{ini} \in N$ *and a final vertex* $n_{fin} \in N$*.*

The edge relation is sometimes called the flow*. The reversed edge relation to a given flow graph is called* reverse flow *and denoted by* E_R*.*

Since we allow `SKIP` statements in the WHILE language, we can assume without loss of generality that every flow graph has isolated entries. This means that for an edge relation E and an initial node n_{ini}:

$$\forall n \in N \, : \, (n, n_{ini}) \notin E.$$

In the same way we assume that every flow graph has isolated exits which means that for an edge relation E and a final node n_{fin}:

$$\forall n \in N \, : \, (n_{fin}, n) \notin E.$$

These assumptions make things technically easier when doing data flow analysis and are generally assumed in standard literature.

Figure 3.7 shows the corresponding data flow graph for the example program of Figure 3.4. The elementary blocks are number as is Figure 3.6.

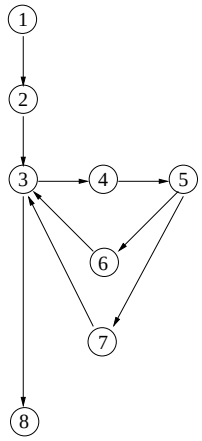

Figure 3.7: A data flow graph

3.4.2 Classical Data Flow Examples

In order to illustrate data flow analysis techniques and what they are capable of, we present a number of classical analysis problems and relate these to flow graphs. This section is mainly targeted to give some intuitive understanding about the class of problems and how these are solved. A formal framework and algorithms for actually solving these problems are given in the subsequent sections.

Available Expression Analysis

The problem to be solved for available expression analysis can be stated as follows:

> For all program nodes, which expressions must have already been computed, and not modified, on all paths to the program point?

This question originates from compiler construction and the answer to it is useful for avoiding re-computation of an expression. A bit more formal the problem can be rephrased as:

Definition 3.17 (available expressions problem)
Determine for every node n of a program P the set of expression Expr_P such that in every path from the initial node n_{ini} to n all expression $e \in \mathsf{Expr}_P$ are computed and not modified prior to n.

Let us consider the labeled program of Figure 3.6. Obviously, at label 2 the expression `z+1` is available, since it has just been computed in the preceding assignment. Moreover, the variable `z` is never redefined in the whole program, this means that for all program points the expression `z+1` is available as well. On the other hand, the expression `z+z` is at no other point available than at the exit of label 6, right after it has been computed. For all other labels, e.g., at the entry of label 4, the expression `z+z` *might* be still available after taking the then-branch of the conditional, but it is not ensured that it *must* been available as demanded in the definition, since on the first entry to the loop this expression is definitely not available and afterwards only if the then-branch is taken. Hence, it is not available on every path leading there.

We list for every program point of the program printed in Figure 3.6 the set of available expressions:

label	entry	exit
1	$\emptyset$	{z+1}
2	{z+1}	{z+1}
3	{z+1}	{z+1}
4	{z+1}	{z+1, x+x}
5	{z+1, x+x}	{z+1, x+x, y*y}
6	{z+1, x+x, y*y}	{z+1, x+x, z+z}
7	{z+1, x+x, y*y}	{z+1, x+x, z*z}
8	{z+1}	{z+1}

Note that it is not sufficient to talk about the available expressions "at label" but one has to distinguish between "at the entry of label" and "at the exit of label", since any label corresponds to some block where expression are generated while others are killed.

Section 3.4.3 and Section 3.4.4 present how the sets of available expressions can be determined for each program point and how they are computed in an algorithmic manner. Here the intuitive idea is the main objective.

Reaching Definitions Analysis

The *reaching definition analysis* is close to the available expressions analysis, but not expressions are examined but assignments and it is a not a *must* reachability but a *may* one. This can be phrased as:

> For all program nodes, which assignments may have already been made, and not overwritten, on all paths to the program point?

The motivation of this analysis is created from the urge to have compact code and to determine relation between blocks that generate assignments and blocks which use them (read the assigned values). This information sometimes helps to reshuffle parts of the code to obtain better results for compiling.

The above description of the reaching definitions problem can be defined as follows:

Definition 3.18 (reaching definitions problem)
Determine for every node n of a program P the set of assignments $\mathcal{A} \subseteq \mathsf{Assign}_P$ such that for any assignments $a \in \mathcal{A}$ there exists a path from the initial node n_{ini} to n where a is computed and not overwritten prior to n.

Let us consider the labeled program of Figure 3.6 once more. The available expressions analysis in the previous section yields that at the exit of program point 2, the only available expressions is `=z+1`. However, the set of assignments that *may* be made without prior overwriting at the entry of 3 is considerably larger: {`x:=z+1`, `y:=z+1`, `x:=x+x`, `y:=z+z`, `y:=z*z`}. Any of these assignments may have been unaltered when reaching the entry of 3, either since it is the first time the loop is entered or one of the if-then-else branches has been taken. Moreover, the whole assignment is of interest and not just the expression on the right hand side as in the previous example. However, within the if-then-else construct it is clear that the assignment `x:=z+1` is not a reaching one, since the assignment in 4 has definitely overwritten it.

We list for every program point of the program in Figure 3.6 the set of reaching definitions, for the sake of brevity we abbreviate any assignment of the form `x:=e` by $(\mathtt{x}, n)$ where n is the number of the label of the corresponding assignment.

label	entry	exit
1	$\emptyset$	$\{(\mathtt{x},1)\}$
2	$\{(\mathtt{x},1)\}$	$\{(\mathtt{x},1),(\mathtt{y},2)\}$
3	$\{(\mathtt{x},1),(\mathtt{x},4),(\mathtt{y},2),(\mathtt{y},6),(\mathtt{y},7)\}$	$\{(\mathtt{x},1),(\mathtt{x},4),(\mathtt{y},2),(\mathtt{y},6),(\mathtt{y},7)\}$
4	$\{(\mathtt{x},1),(\mathtt{x},4),(\mathtt{y},2),(\mathtt{y},6),(\mathtt{y},7)\}$	$\{(\mathtt{x},4),(\mathtt{y},2),(\mathtt{y},6),(\mathtt{y},7)\}$
5	$\{(\mathtt{x},4),(\mathtt{y},2),(\mathtt{y},6),(\mathtt{y},7)\}$	$\{(\mathtt{x},4),(\mathtt{y},2),(\mathtt{y},6),(\mathtt{y},7)\}$
6	$\{(\mathtt{x},4),(\mathtt{y},2),(\mathtt{y},6),(\mathtt{y},7)\}$	$\{(\mathtt{x},4),(\mathtt{y},6)\}$
7	$\{(\mathtt{x},4),(\mathtt{y},2),(\mathtt{y},6),(\mathtt{y},7)\}$	$\{(\mathtt{x},4),(\mathtt{y},7)\}$
8	$\{(\mathtt{x},1),(\mathtt{x},4),(\mathtt{y},2),(\mathtt{y},6),(\mathtt{y},7)\}$	$\{(\mathtt{x},1),(\mathtt{x},4),(\mathtt{y},2),(\mathtt{y},6),(\mathtt{y},7)\}$

As it is observable from this example, the set of reaching definition grows very large if there are many variables which are not assigned a value frequently or if there are many paths, i.e., there is a lot of branching.

Very Busy Expression Analysis

An expression is call *very busy* at the exit of some program point, if this expression is used on all subsequent paths before being overwritten. The goal is to determine:

> For every program point, which expression must be very busy at the exit from that point?

The use of this analysis is obvious, whenever it is known that an expression is very busy, it is for sure that it will be used again and, thus, it value can be stored for later use. This technique is also known as *hoisting* an expression. We define the very busy expression analysis goal as follows:

Definition 3.19 (very busy expression problem)
Determine for every node $n \in N$ of a program P the maximal set of expressions $\mathcal{E} \subseteq \mathsf{Expr}_P$ such that for every expression $e \in \mathcal{E}$ and for every path leaving n, there exists a program point $n_k \in N$ such that e is used in the statement corresponding to n_k and any variable in e is not altered prior to n_k.

We illustrate this definition by Figure 3.6. The only very busy expression available in the example is `z+1` at the exit of block 1, since this expression will definitely be used again in block 2. For all the other expressions it is not guaranteed that they will ever be used at least twice. For instance, any expression in the loop may be used again, by looping once more, but it is not sure that they *must* be used again since the loop might as well terminate without re-using any expression.

For the sake of completeness, we present the full analysis result to the very busy expression problem of the program in Figure 3.6:

label	entry	exit
1	$\emptyset$	{z+1}
2	{z+1}	$\emptyset$
3	$\emptyset$	$\emptyset$
4	$\emptyset$	$\emptyset$
5	$\emptyset$	$\emptyset$
6	$\emptyset$	$\emptyset$
7	$\emptyset$	$\emptyset$
8	$\emptyset$	$\emptyset$

Although in this example there is only one very busy expression, in general there can be significant improvements from storing values that must otherwise be re-computed.

Live Variable Analysis

A variable on is called *live* from the exit of the considered program point, if there exists a path to a use of the variable such that the variable is not overwritten in between. The task of the live variable analysis is to determine:

> For every program point, which variables may be live at the exit from that point?

This analysis appears to be useful for *dead code elimination.* Consider an elementary block assigning some value to a variable v. If from that block onwards the variable is not used anymore or is overwritten before being used, i.e., is not live, the first assignment to v was obviously of no use at all, hence, it can freely be eliminated. Formally, the live variable problem can be phrased as:

Definition 3.20 (live variable problem)
For every program point $n \in N$ of a program P determine the maximal set of variables $\mathcal{V} \subseteq \mathsf{Var}_P$ such that for every variable $v \in \mathcal{V}$ there exists a path from n to some program point $n_k \in N$ where v is used without being altered prior to n_k.

For the example in Figure 3.6 the variable x is live from the exit of block 1 for any program point despite 8 and the exit of 3. The reason is, there exists always a path in the program such that x might be used without being overwritten before. On the other hand, y is, e.g., not live anymore after the exit of 5, since in every possible path from there, i.e., any if-then-else branch, y is overwritten before being used again. However, from entry of 8 onwards no variable is used anymore.

We give the full analysis result of the live variable problem:

label	entry	exit
1	$\emptyset$	$\{\mathtt{x}\}$
2	$\{\mathtt{x}\}$	$\{\mathtt{x},\mathtt{y}\}$
3	$\{\mathtt{x},\mathtt{y}\}$	$\{\mathtt{x},\mathtt{y}\}$
4	$\{\mathtt{x},\mathtt{y}\}$	$\{\mathtt{x},\mathtt{y}\}$
5	$\{\mathtt{x},\mathtt{y}\}$	$\{\mathtt{x}\}$
6	$\{\mathtt{x}\}$	$\{\mathtt{x},\mathtt{y}\}$
7	$\{\mathtt{x}\}$	$\{\mathtt{x},\mathtt{y}\}$
8	$\emptyset$	$\emptyset$

3.4.3 Formal Data Flow Framework

In the previous sections we gave some basic definitions and provided some intuition about different data flow analysis problems. The aim of this section is to put data flow analysis into a formal framework.

First of all data flow analysis requires a framework for representing the properties of the flow problem. Like in available expression analysis there is the need to represent the space of possible expressions or in live variable analysis the space of possible variables for each program point. Moreover, operations and an order on these properties are necessary to compare different intermediate solutions and decide which one approximates better.

Definition 3.21 (property space)
The data flow information and operations on them are represented in a property space $\langle V, \sqsubseteq \rangle$*, which forms a complete lattice.*

As seen in Section 3.4.2 the properties for a given block may differ from the entry to the exit of that block depending on its statements. Transfer functions are used to produce this effect.

Definition 3.22 (transfer function)
A transfer function *on a property space $\langle V, \sqsubseteq \rangle$ if a monotone mapping $f_b : V \to V$ where $b \in \mathsf{Block}$.*

It is natural to demand monotonicity here, since any increase in the input knowledge should definitely not lead to a decrease in the knowledge about the output.

Definition 3.23 (monotone function space)
The monotone function space $\mathcal{F}$ *over a lattice $\langle V, \sqsubseteq \rangle$ is the set of all functions such that:*

1. $\exists id \in \mathcal{F} : \forall v \in V : id(v) = v$,
2. $\forall v \in V : \exists f \in \mathcal{F} : \forall v' \in V : f(v') = v$,
3. $\forall f, g \in \mathcal{F} : f \circ g \in \mathcal{F}$, *and*

4. $\forall f \in \mathcal{F} : \forall v_1, v_2 \in V : f(v_1 \sqcap v_2) \sqsubseteq f(v_1) \sqcap f(v_2)$.

The first requirement in the above definition demands the existence of the identity relation, the second characterizes the zeros of a space, the third the closure under composition and the last states monotonicity.

The combination of a lattice of values and a collection of transfer function yields a description of a class of data flow problems.

Definition 3.24 (monotone data flow framework)
A data flow framework $D = (V, \mathcal{F})$ *consists of a lattice over data flow values* $\langle V, \sqsubseteq \rangle$ *and function space* $\mathcal{F}$ *over it.*

In order to implement a data flow framework, a common requirement is that the lattice $\langle V, \sqsubseteq \rangle$ satisfies the ascending chain condition, i.e., any computation finally stabilizes.

Sometimes more efficient algorithms for a data flow problem can be constructed if a data flow framework is *distributive.*

Definition 3.25 (distributive data flow framework)
A data flow framework $D = (V, \mathcal{F})$ *is* distributive *if it is a monotone data flow frame work and satisfies:*

$$\forall f \in \mathcal{F} : \forall v_1, v_2 \in V : f(v_1 \sqcup v_2) = f(v_1) \sqcup f(v_2).$$

By now we have defined the notions of flow graphs and data flow frameworks, however, there is yet no relation between them. This missing link is given by the following definition:

Definition 3.26 (instance of a data flow framework)
An instance $I = (G, \beta)$ *of a data flow framework* $D = (V, \mathcal{F})$ *is a binding of transfer functions to nodes of the flow graph* $G = (N, E, n_{ini}, n_{fin})$*. This is accomplished through a binding function* $\beta : N \to \mathcal{F}$*.*

The binding function β associates a transfer function $f \in \mathcal{F}$ to every program node $n \in N$. This transfer function captures the effects of that node with respect to the information being gathered throughout the analysis process and which are represented as values in V. For convenience we denote the transfer function $\beta(n)$ by f_n.

Example 3.2 The idea of a data flow framework is illustrated in Figure 3.8. The given data flow graph represents the program of Figure 3.4 without the explicit initial and final nodes. Moreover, it is illustrated for node 2, how the binding function β associates a transfer function f to each node.

The solution of a data flow problem is an approximation to the problem information to each program node in the flow graph. Data flow analysis problems can be formulated directly as equations that give the information

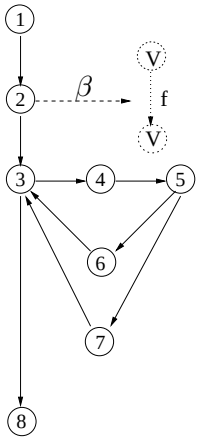

Figure 3.8: Data flow framework

for a node in terms of the information at its predecessor and functions that capture the local effects of a node:

$$out(n) = f_n(in(n_{pred_1}), \ldots, (in_{pred_k}))$$

where $in(n)$ is the problem information at the entry of node $n \in N$ and $out(n)$ is the information at the exit of node n. A data flow framework is a formalism for describing a class of data flow analysis problems. It provides the types for the $in(.)/out(.)$ variables and the transfer function $f \in \mathcal{F}$. An instance of a framework provides information that tells us which variables and functions to equate; thus providing a system of related equations:

$$\begin{aligned} out(n_0) &= \top \\ \forall n \in N - \{n_0\} : out(n) &= \bigsqcap_{i \in Preds(n)} f_i(in(i)) \end{aligned}$$

A data flow analysis problem is *solved* by computing the greatest fixed point of the above equation system. In this equation system n_0 denotes the entry node of the analysis (either n_{ini} or n_{fin}), $\top$ denotes the top element of the lattice V, $\bigsqcap$ the meet operation and $Preds(.)$ the set of predecessor nodes. The actual instances of these notions depend on the analysis problem and whether a forward or a backward directed analysis is performed.

For the classical examples, the instantiations of the analysis scheme is presented in Table 3.4.3. Alternatively, the data flow problem can be stated in terms of a least fixed point if in Table 3.4.3 top is replaced by bottom and the order is reversed.

We go into detail for two examples of the presented table:

Table 3.1: Different analysis schemes

	avail. expr.	reaching def.	very busy expr.	live var.
V	$\mathcal{P}(\mathsf{Expr}_P)$	$\mathcal{P}(\mathsf{Expr}_P \times \mathsf{Block}_P)$	$\mathcal{P}(\mathsf{Expr}_P)$	$\mathcal{P}(\mathsf{Var}_P)$
$\sqsupseteq$	$\supseteq$	$\subseteq$	$\supseteq$	$\subseteq$
$\bigsqcap$	$\bigcap$	$\bigcup$	$\bigcap$	$\bigcup$
$\top$	Expr_P	$\emptyset$	Expr_P	$\emptyset$
flow	E	E	E^R	E^R
n_0	n_{ini}	n_{ini}	n_{fin}	n_{fin}

Example 3.3 For live variables the analysis is usually done in a backwards fashion, i.e., we consider the backward flow E^R of a program P. Starting from the final node n_{fin}, information is propagated backwards in such a manner that every assignment to a variable makes the variable live before this assignment, which is definitely true. For merging points the information is the union of the incoming information, since we are interested in the set of variables which *may* be live at this particular point. The least information we might know, is that no variables are live, i.e., the bottom element is $\emptyset$ and, hence, we consider the sets are partially ordered by $\subseteq$ resulting into the lattice $\langle \mathcal{P}(\mathsf{Var}_P), \subseteq \rangle$.

Example 3.4 In contrast to the live variable analysis the available expression analysis is a *must* analysis, i.e., we are interested in the set of expression which definitely must arrive at some program point. Hence, whenever it comes to merging operations the intersection is used. Moreover, the available expression analysis is done in a forward directed way, i.e., starting from the initial node n_{ini} the flow E is followed. This is clear when considering that if an expression is used we are interested how long it will be available without being overwritten. Since we are always interested in the *least* solution of the equation system, it is natural to define the bottom element as full set Expr and the partial order as $\supseteq$. Hence, the lattice is $\langle \mathcal{P}(\mathsf{Expr}_P), \supseteq \rangle$.

3.4.4 Iterative Solvers for General Frameworks

The formal data flow problem is defined in Section 3.4.3. But there is no algorithmic way presented on how to solve data flow problems such as the ones in Section 3.4.2. In this section we present for each class of forward/backward data flow problems a general algorithm to their solution.

Algorithm 3.9: Iterative solution to general forward directed data flow problems.

Input: A data flow framework $D = (V, \mathcal{F})$ on a flow graph $G = (N, E, n_{ini}, n_{fin})$ with a function $Preds(.)$ to determine the set of predecessor nodes and an extremal value v_{ini} for the initial node.

$out(n_{ini}) = v_{ini}$;
for (each node $n \in N$ other than n_{ini}) **do** $out(n) = \top$ **od**;
$change =$ **true**;
while ($change$) **do**
  $change =$ **false**;
  for (each node $n \in N$ other than n_{ini}) **do**
    $in(n) = \bigsqcap_{n_i \in Preds(n)} out(n_i)$;
    $oldout = out(n)$;
    $out(n) = f_n(in(n))$;
    if $out(n) \neq oldout$ **then** $change =$ **true fi**;
  od
od

Figure 3.9: General Algorithm for forward analysis

Output: The data flow solutions $in(n)$ and $out(n)$ for every node $n \in N$.

Algorithm 3.10: Iterative solution to general backward directed data flow problems.

Input: A data flow framework $D = (V, \mathcal{F})$ on a flow graph $G = (N, E, n_{ini}, n_{fin})$ with a function $Succs(.)$ to determine the set of successor nodes and an extremal value v_{fin} for the final node.

Output: The data flow solutions $in(n)$ and $out(n)$ for every node $n \in N$.

The algorithms for forward and backward problems are very similar. In both algorithms the starting node is initialized and all other nodes get the information as given by the top value. For the forward analysis the entry is the initial node and for the backward analysis the entry is the final node. Next, the transfer functions and meet operations, where necessary, are applied to each node. Depending on the result, i.e, whether anything has changed or not, the whole process is iterated once more until every node is stable, i.e., an application of the transfer functions and meet operations to any node does not change the result. Hence, a fixed point is reached.

Remark 3.1 We can make the following observations:

1. If the algorithms converge, the result is a fixed point solution to the data flow equation system.

2. Given a monotone framework and that all program point are initialized with the $\top$ element. If the algorithm converges it will produce the maximum fixed point (MFP) solution to the equation system.

$out(n_{fin}) = v_{fin}$;
for (each node $n \in N$ other than n_{fin}) **do** $in(n) = \top$ **od**;
$change =$ **true**;
while ($change$) **do**
  $change =$ **false**;
  for (each node $n \in N$ other than n_{fin}) **do**
    $out(n) = \bigsqcap_{n_i \in Succs(n)} in(n_i)$;
    $oldin = in(n)$;
    $in(n) = f_n(out(n))$;
    if $in(n) \neq oldin$ **then** $change =$ **true fi**;
  od
od

Figure 3.10: General algorithm for backward analysis

3. If the data flow problem is based on a lattice of finite hight the algorithms are guaranteed to converge.

Approximations of the Ideal

A valid question to ask is: What is the ideal solution to a data flow problem? The ideal is to take the meet over all paths of a program which are actually executed. This means, every run of a program is covered but nothing more or less. Unfortunately, it is undecidable to determine exactly these paths. An approximation to this is the set of all paths in a program. Certainly, this is an over approximation, since the program semantics is not taken into account and, i.e., more possible executions are considered than there actually exist. For instance, if the condition of the while-loop of Figure 3.6 is set to *false* there will be no execution of any assignment in its body. However, it yields a solution to the data flow problem which is often called the *meet over all paths* (MOP). Note that the set of all paths might still be infinite, e.g., if the flow graph contains cycles. The computation of an MOP solution is, hence, also undecidable.

The maximum fixed point solution does not first compute all paths and afterwards the meet of them. It applies the meet operation whenever it reaches some confluence point and it iterates not necessarily in the order of execution. However, it is only guaranteed to exist under the mentioned circumstances of Remark 3.1. What is it worth then?

Lemma 3.2
Given the instance $I = (G, \beta)$ of a data flow framework $D = (V, \mathcal{F})$. Then:

1. *The MFP solution safely approximates the MOP solution. This means, if the meet operation of V is $\bigcap$ then the MFP solution is a subset of*

the MOP solution and if the meet operation of V is $\bigcup$ then the the MFP solution is a superset of the MOP solution.

2. *If the data flow frame work is distributive the MFP solution equals the MOP solution.*

Proof [NNH99] ∎

Iteration Strategies

The random application of transfer functions to all nodes in each iteration is not very efficient, it is just slightly better than completely randomly choosing nodes and to do transformations on these, which is called *chaotic iteration.* The reason is that these strategies to not take into account any information flow and how information of different nodes interrelate. E.g., in the program of Figure 3.6 information from node 1 will propagate via node 2, via node 3 and so on. It is therefore unnecessary to compute the information of node 7 or 8 right after a start from node 1. More efficient strategies often rely on *worklists.* These represent ordered queues which determine the order of transfer function applications. One task is to find well suited strategies to build and update these worklists. We do not go into detail here, but return to this topic more deeply in Section 5.3.3.

3.5 Abstract Interpretation

Abstractions are key issues when designing, verifying or implementing software. Any model or specification of a system is an abstraction of the real world. Certainly, not every aspect of the real world is covered by a specification since it is based on 'relevant' facts only. The same holds for the analysis of a program or a system specification. Abstractions are done in order to obtain a finite model when, e.g., doing model checking or to reduce the size of the model to relevant facts only. Even for programming abstractions are done since a program does not take care of every aspect of the real world and neglects aspects which are not directly related to the tasks which have to be performed. For instance, it is not necessary to respect input values from certain sensors if these sensors are irrelevant to the control task.

The goal of this section is to provide a notion of abstraction and the interpretation of models or programs on abstract levels. We mainly focus on ideas and notions introduced by Patrick and Radhia Cousot [Cou78], [CC79]. These have been used in various fields and a more complete introduction can be found, e.g., in [Cou81], [CC92], [Cou01] or [NNH99].

3.5.1 Galois Connections

In order to reason formally about abstractions and abstract interpretation let us consider two domains P and Q where P is the *concrete* and Q is the *abstract* domain. To express a relationship between these domains we define a an *abstraction function*

$$\alpha : P \to Q$$

mapping elements of the concrete domain to the abstract domain and we define a *concretion function*

$$\gamma : Q \to P$$

mapping elements from the abstract domain to the concrete one. However, abstractions and concretion should not be arbitrary mappings between two domains but should reflect the intuitive notion of their relation, i.e., that an abstraction is an approximation of something concrete and it should be possible to move from one to the other. Therefore, we define in the spirit of [Ore44]:

Definition 3.27 (Galois Connection)
Given two posets $\langle P, \sqsubseteq_P \rangle$ and $\langle Q, \sqsubseteq_Q \rangle$ with their abstraction function α and concretion function γ. Then, (P, α, γ, Q) forms a Galois connection *if*

$$\forall x \in P \,:\, \forall y \in Q \,:\, \alpha(x) \sqsubseteq_Q y \Leftrightarrow x \sqsubseteq_P \gamma(y).$$

In this work we regard abstractions in terms of Galois connections between two domains. This means, whenever going back and forth between an abstract and a concrete domain we are still safe with respect to any local approximation $\sqsubseteq$ although we might loose precision. This is illustrated in the following example.

Example 3.5 Let $\mathcal{I}$ be the set of all intervals over numbers augmented by the top element $[-\infty, +\infty]$ which denotes the interval comprising all numbers including infinity. The empty interval [] represents the bottom element $\bot$. Consider the poset $\langle \mathcal{P}(\mathbb{Z}), \subseteq \rangle$ which is the power set of numbers with the standard partial order on inclusions. Moreover, consider the poset $\langle \mathcal{I}, \subseteq_{\mathcal{I}} \rangle$ which is the set of intervals with the straightforward notion of inclusion. In fact, both posets are lattices. Let the abstraction function α be determined by

$$\alpha \,:\, z \mapsto [\min z, \max z]$$

mapping every set of numbers to the interval defined by the maximal and minimal element of the set. Conversely, let the concretion function γ be determined by

$$\gamma \,:\, [q_l, q_u] \mapsto \{z \in \mathbb{Z} \mid q_l \leq z \leq q_u\}$$

mapping every interval to the set of numbers covered by the interval.

For a given set $Z = \{-1, 4, 5\}$ of numbers we have $\alpha(Z) = [-1, 5]$ and $\gamma(\alpha(Z)) = \{-1, 0, 1, 2, 3, 4, 5\}$. In particular, $\alpha(Z) \subseteq_{\mathcal{I}} [-1, 5] \Leftrightarrow Z \subseteq \gamma([-1, 5])$. It is easy to see that $(\mathcal{P}(\mathbb{Z}), \alpha, \gamma, \mathcal{I})$ forms a Galois connection since, if the abstraction of a set, i.e., an interval, is included in another interval, the concretion of the later interval yields at least a set that comprises the original one, and vice versa. This coincides well with the natural understanding of abstractions.

Sometimes the interest lies not only in the abstraction of one system into the other one but successive abstractions until a desired level is reached. This requires some means of composition.

Definition 3.28 (Sequential Composition of Galois Connections)
Given the Galois connections $(P, \alpha_1, \gamma_1, Q)$ *and* $(Q, \alpha_2, \gamma_2, R)$*. Their* sequential composition *is defined by* $(P, \alpha_2 \circ \alpha_1, \gamma_1 \circ \gamma_2, R)$*.*

Hence, the sequential composition is based on a sequential application of the abstraction and concretion functions. It is easy to check that the result of this composition is again a Galois connection.

For the remainder of this work we will assume that all the Galois connections do not involve simple posets but complete lattices, if not otherwise stated. Moreover, if we denote a concrete lattice by L then we denote the abstract lattice by $L^{\#}$.

3.5.2 Fixed Point Approximations

As seen in the previous section Galois connections provide means of abstraction for related lattices, i.e., domains. In this section we study the relationship of fixed points on concrete and abstract lattices.

Definition 3.29 (Safe Approximation)
Given a Galois connection $(L, \alpha, \gamma, L^{\#})$ *with the continuous functions* $\phi : L \to L$ *and* $\phi^{\#} : L^{\#} \to L^{\#}$*. We say that* $\phi^{\#}$ *is a* safe approximation ϕ *if*

$$\alpha \circ \phi \circ \gamma \sqsubseteq_{L^{\#}} \phi^{\#}.$$

This means, the operator $\phi^{\#}$ approximates the operator ϕ safely, if on an abstract level $\phi^{\#}$ comprises the effects of ϕ and ensures safety when going back and forth between an abstract and a concrete domain.

Moreover, as shown in [Cou81] for any least fixed point $\mu(\phi)$ its abstract counterpart $\mu(\phi^{\#})$ is also iteratively computable and is a safe approximation, i.e.:

$$\mu(\phi) \sqsubseteq_L \gamma(\mu(\phi^{\#}))$$

or in other words, the concretion of $\mu(\phi^{\#})$ safely approximates the concrete fixed point $\mu(\phi)$.

A question that arises is, how to compute safe approximating operators? By definition, a safe approximation $\phi^\#$ of ϕ is always guaranteed by computing $\phi^\#$ such that $\alpha \circ \phi \circ \gamma = \phi^\#$. However, this is often impractical and might even be uncomputable, since $\gamma(l^\#)$ for $l^\# \in L^\#$ is not necessarily finite. In these cases the abstract operator is constructed manually and later shown to be a safe approximation.

Moreover, we are interested in reaching a fixed point on the abstract level, even though there might be none on the concrete level. Fixed point theory states that we can always effectively compute a fixed point of a monotone function, if its lattice is finite or it satisfies the ascending chain condition [DP90]. However, in practice the lattice might not satisfy either of these requirements or it is of finite hight (i.e., it has only finite chains), but it is still too costly to compute the fixed point. Subsequently, we present acceleration techniques that ensure a safe approximation of abstract fixed points for lattices of very high or even infinite hight.

Widening

As reasoned before, the least fixed point can be computed effectively on a lattice, if the lattice satisfies the ascending chain condition. This means, the chain produced by iterating

$$\begin{aligned} \mu_0^\# &= \bot \\ \mu_{k+1}^\# &= \phi^\#(\mu_k) \end{aligned}$$

where $\phi^\#$ is monotone, satisfies

$$\mu_k^\# \sqsubseteq \mu_{k+1}^\#$$

and will always converge.

The crucial question is, how can we ensure the existence of finite ascending chains? One way to do so is to introduce some acceleration operator that does terminate the iteration sequence (prematurely) by calculating an upper bound which is a post-fixed point, i.e., a safe approximation of the original fixed point.

Definition 3.30 (Widening Operator)
Given a complete lattice $\langle L, \sqsubseteq \rangle$. A sequence $\nabla = (\nabla_n)_{n \in \mathbb{N}}$ of operators is called widening operator*, if every operator ∇_n satisfies*

$$\forall x, y \in L : x \sqcup y \sqsubseteq x \nabla_n y$$

and for every chain $(x_n)_{n\in\mathbb{N}}$ *the chain* $(y_n)_{n\in\mathbb{N}}$ *defined by*

$$\begin{aligned} y_0 &= x_0 \\ y_{n+1} &= y_n \nabla_n x_{n+1} \end{aligned}$$

stabilizes at some point, i.e., there exists a certain $n_k \in \mathbb{N}$ *where*

$$\forall n \in \mathbb{N} : n \geq n_k :\Rightarrow y_n = y_{n_k}.$$

This means the resulting sequence is stable *from a certain sequence element onwards.*

To understand that any fixed point induced by widening is indeed a safe approximation of the original fixed point, consider the following: Given a continuous function $f : L \to L$ in a complete lattice $\langle L, \sqsubseteq\rangle$. Let μ_f denote the least fixed point of f which exists by Lemma 3.1 and Theorem 3.2. This means

$$\mu_f = \bigsqcup_{n\in\mathbb{N}} f^n(\bot).$$

In particular μ_f is an upper bound of the chain

$$\begin{aligned} \mu_{f_0} &= \bot \\ \mu_{f_{n+1}} &= f(\mu_{f_n}). \end{aligned}$$

Assuming a widening operator ∇ we can compute the sequence

$$\begin{aligned} \varphi_0 &= \bot \\ \varphi_{n+1} &= \begin{cases} \varphi_n \nabla_n f(\varphi_n) & \text{if } \neg(f(\varphi_n) \sqsubseteq \varphi_n) \\ \varphi_n & \text{if } f(\varphi_n) \sqsubseteq \varphi_n \end{cases} \end{aligned}$$

and know by definition of the widening operator that there is a least element where the sequence stabilizes, i.e., there exists a least fixed point, μ_f^∇, of the widening sequence. Moreover, testing $f(\varphi_n) \sqsubseteq \varphi_n$ exactly corresponds to testing whether φ_n is an upper approximation of $f(\varphi_n)$. Hence, any fixed point μ_f^∇ is an upper approximation of μ_f. A detailed proof of this can be found in [Bou92].

In particular this leads to the result that in a Galois connection setting $(L, \alpha, \gamma, L^\#)$ any abstract fixed point $\mu(\phi^\#)$ can be safely approximated by its corresponding fixed point $\mu^\nabla(\phi^\#)$ obtained by iterated widening on the abstract level defined as follows:

$$\varphi_0^\# = \bot \tag{3.1}$$

$$\varphi_{n+1}^\# = \varphi_n^\# \nabla_n \phi^\#(\varphi_n^\#). \tag{3.2}$$

Intuitively this means, if we start iterating the abstract fixed point from the bottom element, apply widening in any iteration and use the result for the next iteration. Since widening leads by definition to a converging sequence the equation system determined by 3.1 and 3.2 computes a safe approximation of the least abstract fixed point.

Example 3.6 Consider again the Galois connection $(\mathcal{P}(\mathbb{Z}), \alpha, \gamma, \mathcal{I})$ of Example 3.5. Assume a function

$$f : \mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$$

on the concrete domain such that

$$f : Z \mapsto \begin{cases} \{0\} & \text{if } Z = \emptyset \\ \{z \in \mathbb{Z} \mid \min Z \leq z \leq (\max Z) + 1\} & \text{if } \max Z < 10 \\ \{z \in \mathbb{Z} \mid (\min Z) - 1 \leq z \leq \max Z\} & \text{if } \max Z \geq 10 \end{cases}$$

and a function

$$f^{\#} : \mathcal{I} \to \mathcal{I}$$

on the abstract domain such that

$$f^{\#} : [a, b] \mapsto \begin{cases} [0, 0] & \text{if } [a, b] = [] \\ [a, b + 1] & \text{if } b < 10 \\ [a - 1, b] & \text{if } b \geq 10. \end{cases}$$

We can observe that the least fixed point μ_f of f is $\{z \in \mathbb{Z} \mid z \leq 10\}$ and the least fixed point $\mu_{f^{\#}}$ of $f^{\#}$ is $[-\infty, 10]$. It is straightforward to proof that $f^{\#}$ is a safe approximation of f. However, for none of the two functions we can effectively compute their fixed point by iteration, since this would require infinitely many of them.

We define a widening operator ∇ on the abstract domain by

$$\begin{array}{rcl} [] \nabla x = x \nabla [] & = & x \\ [a_1, b_1] \nabla [a_2, b_2] & = & [\text{if } a_2 < a_1 \text{ then } -\infty \text{ else } a_1, \\ & & \text{if } b_2 > b_1 \text{ then } +\infty \text{ else } b_1]. \end{array}$$

It can be shown that ∇ is in fact a widening operator, however, a very crude one. This means, whenever we do widening on two intervals and detect that the second interval has higher upper bound or lower lower bound we jump to the respective limit immediately. We use the widening operator ∇ to compute the least fixed point $\mu^{\nabla}_{f^{\#}}$ as follows:

$$\begin{array}{rcllcl} \varphi^{\#}_0 & = & [] & & & \\ \varphi^{\#}_1 & = & [] \nabla_0 f^{\#}([]) & & = & [0, 0] \\ \varphi^{\#}_2 & = & [0, 0] \nabla_1 f^{\#}([0, 0]) & & = & [0, +\infty] \\ \varphi^{\#}_3 & = & [0, +\infty] \nabla_2 f^{\#}([0, +\infty]) & & = & [-\infty, +\infty] \\ \mu^{\nabla}_{f^{\#}} = \varphi^{\#}_4 & = & [-\infty, +\infty] \nabla_3 f^{\#}([-\infty, +\infty]) & & = & [-\infty, +\infty]. \end{array}$$

After only 3 iterations we obtain the least fixed point of the widening sequence, $\mu_{f\#}^{\nabla} = [-\infty, +\infty]$, which is a safe approximation of $\mu_{f\#}$, however, not a very precise one. Note, though, that slightly changing the threshold in $f^{\#}$ (and f) from 10 to 0 would yield the precise fixed point. In the same way the widening operator can be adapted to the threshold 10 such that in the above iteration process the precise fixed point is reached.

Unfortunately, there is always a trade off between quick termination of the iteration process and high precision. Of course there is no algorithmic way to determine the optimal widening operator with respect to the precise fixed point, since otherwise it would be computable right away.

Narrowing

As seen above the use of an acceleration technique to enforce stabilization might lead to great loss of precision. In oder to reduce this gross over-approximation we introduce a second acceleration operator called *narrowing*. The general idea is to start from the approximated fixed point obtained in the widening phase and then to decrease towards the least fixed point but without ever passing below it. Therefore, we define a narrowing operator as follows:

Definition 3.31 (Narrowing Operator)
Given a complete lattice $\langle L, \sqsubseteq \rangle$. A sequence $\Delta = (\Delta_n)_{n\in\mathbb{N}}$ of operators is called widening operator*, if every operator Δ_n satisfies*

$$\forall x, y \in L : y \sqsubseteq x \quad \Rightarrow \quad y \sqsubseteq x \,\Delta_n\, y \sqsubseteq x$$

and for every chain $(y_n)_{n\in\mathbb{N}}$ the chain $(z_n)_{n\in\mathbb{N}}$ defined by

$$\begin{aligned} z_0 &= y_0 \\ z_{n+1} &= z_n \,\Delta_n\, y_{n+1} \end{aligned}$$

stabilizes at a certain $n_k \in \mathbb{N}$:

$$\forall n \in \mathbb{N} : n \geq n_k :\Rightarrow z_n = z_{n_k}.$$

Example 3.7 We continue with Example 3.6. We define a narrowing operator Δ on the abstract domain by

$$\begin{aligned} [] \,\Delta\, x = x \,\Delta\, [] &= [] \\ [a_1, b_1] \,\Delta\, [a_2, b_2] &= [\text{if } a_1 = -\infty \text{ then } a_2 \text{ else } \min(a_1, a_2), \\ &\quad \text{if } b_2 = +\infty \text{ then } b_2 \text{ else } \max(b_1, b_2)]. \end{aligned}$$

This means the Δ operator gives an approximation better than infinity whenever possible and, otherwise, returns the interval comprising exactly both arguments. As mentioned before, the key idea is to apply the narrowing operator to the abstract fixed point obtained through widening. This means, we start from of $[-\infty, +\infty]$ and obtain:

$$\begin{array}{lcl} \psi_0^\# & = & [-\infty, +\infty] \\ \mu^\Delta_{f^\#} = \psi_1^\# & = & [-\infty, +\infty]\ \Delta_0\ f^\#([-\infty, +\infty]) \quad = \quad [-\infty, +\infty] \end{array}$$

The least fixed point $\mu^\Delta_{f^\#}$ obtained through narrowing remains the same we already obtained in the widening phase. But let us consider a function $f^\#_{10}$ defined slightly different than $f^\#$:

$$f^\#_{10} : [a, b] \mapsto \begin{cases} [0, 0] & \text{if } [a, b] = [] \\ [a, b + 1] & \text{if } b < 10 \\ [a - 1, 10] & \text{if } b \geq 10. \end{cases}$$

The only difference to the original function is that we explicitly give the upper limit of the resulting interval (i.e., 10) in the case that b is greater or equal 10. The abstract fixed point is still $[-\infty, 10]$ and by fixed point iteration using widening we still obtain $[-\infty, +\infty]$ as a fixed point. However, using narrowing we now obtain

$$\begin{array}{lcl} \psi_0^\# & = & [-\infty, +\infty] \\ \psi_1^\# & = & [-\infty, +\infty]\ \Delta_0\ f^\#_{10}([-\infty, +\infty]) \quad = \quad [-\infty, 10] \\ \mu^\Delta_{f^\#_{10}} = \psi_2^\# & = & [-\infty, 10]\ \Delta_1\ f^\#_{10}([-\infty, 10]) \quad = \quad [-\infty, 10] \end{array}$$

which correspond to the least fixed point of $f^\#$.

As a conclusion we can draw that widening/narrowing allows us always to effectively compute a fixed point. The design of the widening/narrowing operator as well as the function itself determines the precision of the results. Generally, the function itself cannot be changed and the widening/narrowing is difficult to design such that it yields high precision. Hence, the main contribution of the widening/narrowing operator is to find any fixed point, precision might be enhanced by additional methods. We return to this issue in Section 5.3.4.

3.5.3 Abstract Interpretation for Program Analysis

Abstract interpretation is a very general framework that can be applied to various fields: To syntax in order to compare grammars, to semantics that helps to design semantic hierarchies, to typing, to model checking and program transformations in order to provide suitable means of abstractions.

In this work we are mainly interested in abstract interpretation for program analysis.

In general, to prove any property, e.g., of the WHILE language we introduced in Section 3.4.1 in an automated fashion is impossible due to undecidability. But as we have seen in the previous section, abstract interpretation including widening/narrowing enables us to effectively compute approximated fixed points even though the concrete ones cannot be computed. Thus, the basic idea is to effectively compute approximations of the program semantics and prove properties on this approximation. This enables the detection of approximated run-time errors at compile time. In particular, there are two issues we have to consider: What kind of semantics serves best for this purpose and how should the abstract domain be designed?

Collecting Semantics

A very general and low level approach to represent operational semantics are *traces*. Traces correspond to possible infinite sequences of states. For instance, every run in a Kripke structure as defined in Section 3.3.2 is a trace. As already remarked by Floyd [Flo67], to prove static properties of programs it is sufficient to consider *sets of states* associated with each program point. To illustrate this, consider the program in Figure 3.11 annotated by labels as in Section 3.4.1:

```
[x:=1]1
WHILE [x<10]2 DO
  [x:=x+1]3
OD
[SKIP]4
```

Figure 3.11: Program fragment to illustrate semantics

A trace based semantics for this fragment yields a sequence (only state changes recorded):

$$\langle (x,\bot),(x,1),(x,2),(x,3),(x,4),(x,5),(x,6),(x,7),(x,8,)(x,9,)(x,10)\rangle$$

while the *collecting semantics* describes the sets of possible variable values for each program point. In a data flow analysis like notation we obtain:

label	entry	exit
1	(x,$\emptyset$)	(x,{1})
2	(x,{1,...,9})	(x,{1,...,9})
3	(x,{1,...,9})	(x,{1,...,10})
4	(x,{1,...,10})	(x,{1,...,10})

In this table the entry row covers the possible program variable values before a program point and the exit row after that program point, i.e., after executing the corresponding statement in the given semantics. More formally:

Definition 3.32 (Collecting Semantics)
Consider a concrete domain $\mathcal{D}$. The collecting semantics $\{\!|p|\!\} \in \mathcal{P}(\mathcal{D})$ *of a program p is the set $\{[\![p]\!]_\iota \mid \iota \in \mathsf{In}\}$ of possible output values in $\mathcal{D}$ corresponding to a given set of input values* Block *as defined by the standard semantics $[\![p]\!]$.*

When analyzing program in an abstract interpretation framework, we often ask question such as "Is it possible that x is zero at a specific program point?" or "Does x ever exceed 10 at this program point"? We call questions (and answers) like these *concrete program properties.* On an abstract level we might ask similar questions about, e.g., intervals. However, in order to make full use of the abstract interpretation framework it is necessary to relate the concrete property space $P = \mathcal{P}(\mathcal{D})$ in the collecting semantics of domain $\mathcal{D}$ with an abstract property space $P^\#$ by a Galois connection $(P, \alpha, \gamma, P^\#)$. In fact, the previous examples on $(\mathcal{P}(\mathbb{Z}), \alpha, \gamma, \mathcal{I})$ are just instances of this idea. We return to this in Section 5.3.2.

Abstract Domains

Up to now we focused on intervals as abstract domains only. Although we will use intervals for the remainder of this work, too, it is worth considering other representations for abstract domains. The general idea is to find domains that are most suitable for an abstraction of concrete domains and the representation of the respective program properties to be examined. Finite concrete domains are often less a problem since their analysis reduces to finite state model checking. However, the domains may still be too large for efficient analysis. More difficult are infinite domains like integers.

In general, abstractions can be classified as relational and non-relational. Relational abstractions take relationships between different variables like $x = 2y$ into account, while non-relational abstraction neglect these relationships and consider variables as single entities. For instance, ordinary interval abstractions cover no information about the relationships of different variables. The same holds for the well known sign abstraction [CC79], where values are classified in two categories whether they are negative or not. The same holds, e.g., for simple congruence abstractions [Gra89].

On the other hand structures like octagons or, more general, polyhedra [CH78] approximate the concrete space incorporating relationships between variables. Sophisticated methods take also linear [Gra91] or trapezoid linear congruences [Mas92] into account, which roughly resemble ideas from vector spaces and spans.

It is not clear which abstraction method is optimal for a given class of problems, but generally the more sophisticated representations require also more computation to check for inclusion or to perform unions of such elements. Since we are in particular interested in range checking and giving an upper and lower bound to possible variable values we remain focused on intervals.

Chapter 4

Semantics

4.1 Introduction

PLC programming languages are defined in [IEC98], but this standard is mainly concerned with syntactic and technical issues. The semantics is described in an informal way with leaves plenty of room for different interpretations. Moreover, the described behavior of the PLC programming languages is often ambiguous or incomplete. Since any precise reasoning about PLC programs requires a precise semantics, we define a formal semantics for SFCs and IL in this chapter.

First we consider different modeling issues of PLCs and PLC languages in Section 4.2. This serves as a basis for the design decisions we make about the semantics we introduce subsequently. In Section 4.3 we give a comprehensive introduction to SFCs. We point to ambiguities of the informal semantics as described in the standard and explain how to resolve them. Based of these consideration we define a formal syntax and semantics for SFCs. Moreover, the semantics is parametric, i.e., it can be easily adapted to various interpretations of the semantics as used by existing tools and, therefore, is a unification of existing implementations. Section 4.4 gives an introduction to the language IL. The syntax is defined and we give an SOS style semantics for this language.

4.2 Modeling PLCs

When designing or modeling PLCs there are a number of different views and different level of abstractions possible. This is even more true, if there is an interest in verification where the type of properties plays an important additional role. In this section we describe some possible views and categories that help to understand the different approaches in modeling and verification.

4.2.1 PLCs and their Environment

PLC models and their verification can either concern the composition of a PLC and its environment, or a PLC alone. In the latter case the PLC is modeled in isolation of the environment it is embedded in, this means, the reactive behavior is either built into the PLC model implicitly by the integration of assumptions over the environment, it is neglected, or the PLC is modeled as an open system allowing arbitrary environment behavior.

A PLC model can also be derived solely from the program code or its specification. In principle, all the rich theory about modeling programming languages can be applied here. PLCs use mainly domain specific languages (apart from more recent developments where also languages like C are introduced to PLCs). The advantage of such domain specific languages with restricted functionality is that they allow tailored, optimal models. The disadvantage is, however, that the PLC languages actually used are extremely diverse and often do not reflect the state of the art in programming languages.

On the other hand, to prove properties about the whole system behavior it is desirable to consider both, the PLC and the environment, respectively their models. With respect to modeling one of the relevant questions is how to compose the model of the controller and the model of the environment. This means, their interaction and communication is a crucial modeling aspect. Moreover, also for the environment one has to find a suitable level of abstraction. Clearly, nobody wants to model the whole world around the PLC but relevant parts only.

4.2.2 Scan Cycles

As we have seen in Section 1.4, one of the features that distinguishes PLCs from other systems is that everything is bound to a scan cycle. There are a number of possible ways to model or abstract from this cyclic behavior.

Explicit Scan Cycles

By explicit modeling of the scan cycle we understand that the model includes the cyclic execution mechanism of the PLC program together with its timing information, i.e., its execution times on a machine.

For modern PLCs it is the case that the scan cycle time depends on the actual execution path of the program. A lower bound for a scan cycle is always the (constant) time that the i/o phase takes, together with the amount of operating system actions performed periodically (self-checks etc.). The rest of the scan cycle time consists of the program execution time, which can vary depending on the state of the program (i.e., internal variables and input variables) and the code to be executed.

An explicit scan cycle model often leads to the analysis of timing behavior and worst case execution times. Consider, for instance, a PLC controlled chemical plant: opening and closing a valve and starting or stopping the flow of some liquid takes generally far more time than a scan cycle. Timing is not that critical. On the other hand, "fast" applications as, for instance, air bags have reaction times close to the scan cycle time. In fact, for these applications it has to be shown that the scan cycle time is short enough to solve the control problem adequately (tight loop control). In some environments both phenomena can occur, a generally "slow" plant can contain "fast" subparts. A timing analysis can be crucial.

Another example where an explicit timing model is useful comes with asynchronous communication, where one PLC sends a signal which has to be read by another PLC. The sending PLC must ensure that the signal is stable long enough to be read by the receiving PLC. For the sending PLC, the lower time bound is relevant, and for the receiving its upper time bound of the scan cycle length. Of course, two different PLCs have independent scan cycles.

Implicit Scan Cycles

By an implicit modeling of the scan cycle we understand that the cyclic behavior for program execution is modeled, but the (real-time) duration of each scan cycle is not considered. This abstraction is useful if there is no need for a worst-case timing analysis or reaction times comply well to the synchrony hypothesis anyway.

We consider the following examples: For older PLCs each program execution takes exactly the same time. There is a program memory for a fixed number of instructions that are all executed in each scan cycle. For shorter programs the remaining space is filled with skip-instructions. The worst-case execution time is *fixed.* In some modern PLCs a similar behavior can be found: an alternative program execution mode to the cyclic one as described above is a periodical execution mode. There, the program execution is started periodically with fixed time intervals (where the interval should be longer than one program execution). Finally, there are PLCs with an builtin *time out,* i.e., the cycle lengths might vary, but a worst-case time is pre-defined.

For these examples the scan cycles can be assumed implicitly, e.g., by a *clock tick.* The cycle time itself is irrelevant, what is of interest is that there *is* a cycle and that these cycles can be distinguished.

Abstracted Scan Cycles

By abstracting from scan cycles we understand models where i/o-phase and program execution take place in zero time and is generally not modeled at

all. This type of modeling is often sufficient where one is concerned about program analysis only and not in any timing behavior at all. Generally, these types of models to not include the environment or allow a chaotic environment, i.e., where any input can occur anytime.

Without further assumptions zero execution time would allow for Zeno-behavior, i.e., infinitely many program executions within a finite time interval. Therefore, any combination of a timed model and a model where scan cycles are abstracted has to be chosen carefully.

4.2.3 Time

One classification of PLC models closely related to the other categories depends on the use of time and timers. Time can be incorporated in different ways. Each instruction may take some time or each scan cycle may take time. This notion of time can be based on real numbers or just on discrete clock ticks.

The most precise way of modeling the real time behavior of the program execution is to include the different program execution times explicitly into the model. One possibility to do this is to model each program instruction individually and attaching an explicit duration to it. A run of the model then reassembles not only the concrete, actual program execution instruction-wise but also sums up their execution times. Another possibility, which is less precise, is to define an upper and a lower bound for a whole program execution and letting the program model non-deterministically execute within the time interval defined.

Timing information is also suitable for analysis of delays that are caused by the cyclic operation mode. For example, when analyzing an alarm clock that ideally is started at some moment and should ring after a specified delay, the question is whether this alarm does really ring after the exact amount of time. In a PLC setting it almost never does. The reason is that in general the cyclic behavior induces some additional delay. Therefore, one has to be careful when testing for equality to a point in time or if delays in a program are assumed to be exact.

Another aspect where time is introduced is given by the PLC programming languages themselves. They allow the use of timers and to reason about them.

In general, for approaches including real-time the models generated are quickly getting very large, and further abstraction or sophisticated verification techniques have to be used for analysis.

4.2.4 Software Features

Next to the different programming languages for PLCs as sketched in Section 1.4 there are number a features specific to PLCs.

Datatypes

In the standard PLC languages types such as Booleans, integers and reals are available. For many control tasks reals are not necessary: if some sensor data reaches a threshold then some actuators are set on or off. Moreover, real number computations often cost much time and there is a trend to remove costly real number computations from a PLC to a PC (that communicates with the PLC) in order to keep the scan cycle within predictable bounds. Therefore, it is reasonable to investigate the class of programs where the only data-types are Booleans and Integers.

Language Fragments

PLCs are built by many different manufacturers. Many of them used to define their own programming languages. One intention of the standard was to bring conformity to the confusing variety of languages. The standard contains the five different languages as introduced in Section 1.4.

Besides providing conformity, a standard should also aim to be accepted by a large number of manufacturers. This happens only, if the difference between their traditional programming environments and the ones provided by the standard is not too big. Therefore, the standard has also a collecting function. A sensible restriction of language constructs and a clear definition of semantics of the languages would simply throw several brands out of the game. As a consequence, these topics are on purpose not addressed by the standard. However, this is a disadvantage for a formal treatment of the languages. From the theoretician's point of view the languages are overloaded with constructions that are not at all necessary for expressiveness. Instead, some of them introduce non-trivial semantic ambiguities. On the other hand, there is a precise but unimplementable semantics. An example are the timers: there is a precise definition based on a continuous data-flow model. It *cannot* and *does not* translate to any implementation as specified, because every PLC only approximates a continuous control.

In this situation our conclusions are:

- It does not make sense to give semantics to a full language.
- When trying to formally analyze PLC programs it is essential to carefully select the fragments of languages that are treated.
- It is useful to develop programming guidelines and a programming discipline which restrict the class of programs that should be verified. Even in a defined language fragment it is not desirable to be able to verify every thinkable (and possibly unstructured) program.

However, when considering only language fragments, this should be in the first place a horizontal restriction. It is certainly necessary to integrate

different languages into one formal approach: many PLC programs are (and have to be) written in more than one language. In a typical case, a program structure is given in SFC, the transition conditions are Boolean expressions written in ST, and the actions attached to each step may be in IL.

Timers

Basically, there are two reasons to use timers. They can be characterized as follows:

1. The duration of an *output signal* is controlled, i.e., an output signal has to be stable for a certain time. A simple example here is a flashing signal indicating a critical process for the operator.

2. A timer as a substitute for incomplete knowledge about the environment. This knowledge might also be provided by extra sensors. I.e., a timer substitutes an *input signal*. This is certainly the most important use of timers. E.g., in a chemical plant a valve is opened and the process continues after the tank is empty; either a sensor can indicate that the tank is empty (closed loop control) or experience tells that after some time the tank is certainly empty and the process just waits for that time before it continues (open loop control). In this sense a level sensor can be substituted by a timer. This solution is often chosen, when sensors are too expensive or not available.

   This observation is also applicable for other cases: in protocols timeouts can substitute the knowledge that, e.g., the communication partner has not received a message.

For both classes there are applications where the use of timers can be avoided. In the first case it can be sufficient to count scan cycles and multiply them by the lower cycle time bound, in the second case the controlled environment may be fully equipped with sensors and timers are not needed. [1]

It is obvious that models omitting timers can be useful for a big class of applications. Furthermore, the intended correctness properties are also relevant: many properties are time-independent.

Additional Features

Initially, PLCs were characterized by a very easy and also stable operation mode. Today all sorts of modern features also can be found in PLCs: multi-tasking, interrupts and event-handling, watchdogs, various execution

[1] However, they can be useful for error detection by comparing expected processing time and sensor data.

modes and more. From the theoretical point of view, these features increase complexity in a way that makes formal analysis difficult. The natural consequence is to restrict the applications to those not making use of "complicated" features. E.g., a non-restricted number of interrupts can increase the scan-cycle length in a way that the cycle length gets unpredictable. In extreme cases they can lead to an alarm of a watchdog leading to other interrupts which altogether makes the correct behavior of the controller difficult to guarantee.

On the other hand, there may be cases where interrupts are necessary: e.g., the unfolding of the air-bag in a car has to happen as soon as possible, without delay of even one scan cycle. In such a case interrupts are certainly sensible, and can be treated formally, if we allow only for a restricted number of interrupts within one scan cycle.

Therefore, in order to come closer to real applications also more complex features have to be investigated. Here, the same holds as for the programming languages: the features used should be well-chosen and restricted in their application.

4.2.5 Our Models

As mentioned already in the introduction of this work, parts of this thesis are motivated by struggling in the verification of full hybrid systems, i.e, PLCs and their environment. As a consequence we focus on the most crucial aspect: the software. In this work we focus on two distinct PLC languages and their program verification, namely, SFCs and IL programs. In order to reduce complexity as much as possible we restrict ourselves to the following models:

The PLC is modeled without an explicit environment, we rather model an open system. This means, we assume that the environment may behave chaoticly, i.e., every possible reaction may occur and we have to take this behavior into considerations.

In the verification process we abstract from the use of explicit time and timers. Time always increases complexity significantly and we strive for a system describing PLCs and their software as simple as possible. However, we suggest extensions from untimed to timed SFCs, modeled as *linear hybrid systems.* These allow to reason about time and timers.

For both semantics, IL and SFCs, we use an implicit scan cycle approach. This means, e.g., we can reason about what is happening in a cycle or in a future one, but we do not have any concrete timing information of how long a cycle or a set of instructions takes.

For illustration purposes and to focus on the main features we generally consider restricted sub-classes of each language. However, we cover most concepts which are unique to a language or to PLCs. E.g., SFC concepts such as hierarchy, parallelism or activity manipulation are fully covered while

other less unique concepts such as record data types, exception handling or external function calls are neglected.

4.3 Sequential Function Charts

4.3.1 Introduction

Sequential function charts are defined in [IEC98] as elements of a graphical programming and structuring language for programmable logic controllers. The SFC definitions in the standard IEC 61131-3 are based on IEC 60848 [IEC92], which defines the specification language Grafcet. Grafcet in turn is strongly related to Petri nets [DA92].

Basically, SFCs are transition systems consisting of *steps* (the locations) and *transitions*. For every SFC there exists exactly one *initial step*. Every transition is labeled by an associated transition condition, called *guard*. Moreover, one or more *actions* may be associated to each step. Actions are again SFCs or programs in one of the other programming languages proposed by the standard. Since the actions associated to steps can be SFCs themselves, a concept of hierarchy is provided. An example of a sequential function chart is depicted in Fig. 4.1.

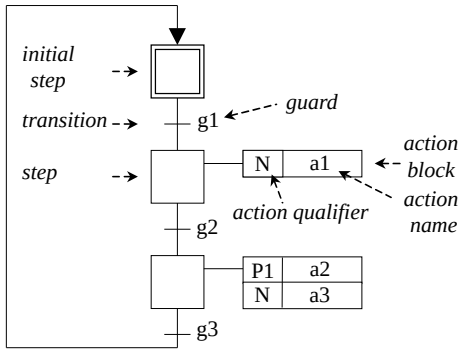

Figure 4.1: Elements of SFCs

The *action blocks* shown in Fig. 4.1 are a graphical means to associate actions to steps. An action block consists of an *action qualifier*, which can be used to specify the activity of the respective action, and the *action name*. Concerning the action qualifiers defined in the standard, we concentrate on those without an associated duration of time. These are the following:

- `N` – ***N****on-stored*
- `R` – ***R****eset*
- `S` – ***S****et or* ***S****tored*

- `P1` – ***P**ulse – rising edge* (activation of corresponding step)
- `P0` – ***P**ulse – falling edge* (deactivation of corresponding step)

Intuitively, the non-stored actions are always active while control resides in the corresponding step, i.e., it is active. In contrast the stored actions remain active even outside their step of activation until the corresponding reset action is called. The actions with the pulse qualifier are performed only once when entering (`P1`) or exiting (`P0`) a step, i.e., when the step gets activated or deactivated. Note, we consider the qualifiers `P1` and `P0`, as introduced in [IEC98] instead of the `P` qualifier, which was defined in the first edition of the standard and we neglect the rather non-intuitive and optional *final scan* behavior which means that each non-Boolean action has to be carried out a further time after being deactivated.

An SFC does not necessarily have to be a single sequence of steps and transitions. For SFCs we can identify a number of different transition types (cf. Figure 4.2). Transition (1) denotes a *simple transition* between two steps, (2) describes *alternative branching*, i.e., the choice between several transitions, (3) *divergence*, (4) *convergence*, and (5) a simultaneity of both of the latter. In this context divergence means a parallel branching from one step to a set of next steps while convergence means the synchronization of several steps in parallel to a single one. Furthermore, direct combinations of both as depicted in (5) are allowed. Moreover, the standard refers to transition like the one guarded by g_3 in Figure 4.1 as *loops*. From our point of view loops as well as alternative branching belong to the same class, namely (sets of) simple transitions.

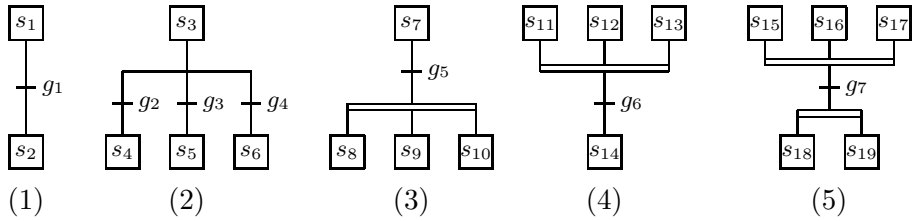

Figure 4.2: Basic transition types.

An additional feature of SFCs is to explicitly assign priorities to alternative branches. This means, when firing transitions the one which has the highest priority among the enabled ones will be taken. If there are no priorities given there often is the implicit rule that the transitions are ordered "from left to right" in decreasing priority.

These basic transition types can be combined into more complex transition structures like in Figure 4.1 and 4.3. However, there are various combinations which do not seem to make sense, e.g., in the SFC shown in Figure 4.3. There we have the transition from s_4 to s_1 which jumps out of a

parallel branch, the converging transition from s_2 to s_5, which jumps from one simultaneous branch to another, and the transition from s_5 and s_6 to s_7, which is a convergence of simultaneous sequences of two steps which are part of an alternative branching starting in step s_3.

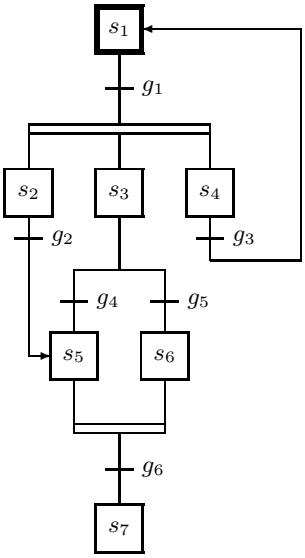

Figure 4.3: Unsafe SFC

Although the standard forbids to use such kinds of "unsafe" or "unreachable" SFCs, it does not give a precise characterization of this phenomenon. For the time being we do not want to pose any explicit restrictions on the SFC syntax. Mainly, because we want to allow as much freedom as possible and because there are commercial tools available which "support" SFCs with these kinds of transitions and do simulations on them. Hence, in the following we define a semantics for sequential function charts which can also cope with such constructions. However, we return to this phenomenon in Section 5.2.3 where we give a characterization of a safe SFC and propose a method to decide this property.

4.3.2 Ambiguities in the Semantics

The standard defines rules for building an SFC from the aforementioned basic elements and describes how to execute SFCs by giving evolution rules similar to the firing rules of Petri nets. However, execution is only defined on an abstract level not taking into account concrete aspects of program execution. As SFC is a *programming language* the exact semantics of execution is of interest and it is therefore important to consider the cyclic SFC execution on a PLC. In every scan cycle first the new input from the environment (i.e., from the sensors of a plant such as pressure or temperature

sensors) is read and stored. Then the PLC program is executed based on the stored input, i.e., the actions of the active steps are executed, which may change the output, and afterwards the transitions are taken. At the end of each cycle the output is sent to the environment, i.e., to the actuators of a plant such as valves and motors.

Although the semantics of SFCs on an abstract level seems to be quite simple, the exact semantics on an operational level is sometimes far from obvious. Let us have a close look at some key issues:

In which order does the firing of transitions and execution of actions take place?

Does the PLC, after reading the inputs, fire the enabled transitions first and then executes the actions or is this done the other way round? This order is important for the initial step. If in the first PLC cycle (or the first time the SFC is executed) the transition following the initial step is already enabled, its actions will only be executed if the order is first executing actions and then firing the transitions. Furthermore, the order is important since the execution of actions might affect the guards of transitions. Consider for instance SFC_1 in Fig. 4.5. Assume control resides in location s_{10} and g_{11} is $z \neq 1$ then the order is obviously important. If the actions are executed first, the transition guarded by g_{11} will never be taken whereas if the transition might fire first this is not guaranteed.

How to deal with parallelism?

In which order are actions of parallel steps executed? Is there any order at all or do we have to cope with non-determinism? As illustrated by the example in Fig. 4.4 the order of execution makes a difference if the actions modify the same variables. Depending on whether executing a_1 before a_2 or not, the transition guarded with $y = 2$ is enabled or not.

The execution order is also not clear if we have more than one action associated to the same step. Are they executed from top to bottom or according to a different rule?

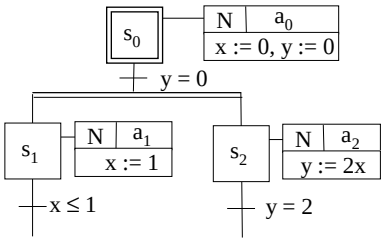

Figure 4.4: In which order are the actions a_1 and a_2 executed?

How to deal with hierarchy of SFCs?

Figure 4.5 shows a top level SFC SFC_0 with a second SFC nested in step s_1 (SFC_1). Has the top level SFC higher priority, that is, are the actions of the top level SFC executed prior to the actions of the nested SFC? If we enter s_1 in SFC_0, does the execution order of actions depend on the hierarchy, i.e., do we first execute a_1 and then the actions a_2 and a_3 of SFC_1? Can SFC_1 reset a_2 or has the top SFC priority and a_2 is executed? Which step of the nested SFC will be entered if it is called again? For instance, if we enter step s_1 again, do we always activate s_{10} or is there a notion of history and the last active step of SFC_1 becomes activated?

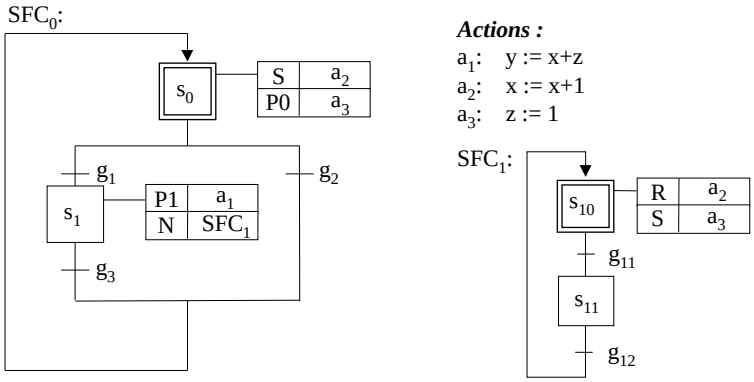

Figure 4.5: How to deal with hierarchy?

Possible Answers to Ambiguities

The standard does not provide any explicit answer to these questions. For the execution order of actions a comparison of different SFC programming tools [BT01] yielded that there exist various orders on actions, e. g., for one tool actions are executed according the alphabetic order of action names, for another one by user pre-defined orders and so on. However, in no case there is non-determinism. This seems to be reasonable since PLCs are in general deterministic. The semantics defined in the following copes with these different semantics by introducing a parameter for the order of actions. Another parameter is introduced for the order of transitions as the standard allows priorities if more than one transition of an alternative branching is enabled.

For the execution order of firing transitions and executing actions we decide that first all actions are executed and then the enabled transitions are fired. This appears to be in line with examined tools. The same holds for hierarchical SFCs where we define that there a notion of history, such that the SFC re-enters into the last active step of the previous execution.

Finally, there is the question whether it is possible to create algebraic loops, e.g., is it possible that a nested SFC calls again the top level SFC with a different qualifier? In general this is not forbidden, but in our framework we assume systems without algebraic loops and leave it to the program designer to avoid these.

The given decisions are the basis for a clear non-ambiguous syntax and semantics of SFCs which is presented in the following.

4.3.3 Syntax of SFCs with Actions

SFCs (or PLC programs in general) have different types of variables, such as input variables, output variables, local variables. The values of the variables may belong to different data types such as Boolean or integer. The valid variable and data types are regulated by the standard IEC 61131-3, already mentioned above. We abstract from these different variable and data types simply saying an SFC has variables which may have different values. To describe the values of all variables we use the notion of a *state*, which is a function assigning a value to each variable.

Definition 4.1 (State)
Let Σ be a nonempty set of elements σ, called states.

A state (i. e., the values of the variables) can be modified by *state transformations.* In [IEC98] different types of programming languages are defined for such state transformations. We abstract from these types by simply saying that a state transformation is a function that transforms a given state into another state. Here state transformations are deterministic; in non-deterministic languages a state is transformed into one of several possible successor states.

Definition 4.2 (State transformation)
A state transformation *is a function $f : \Sigma \to \Sigma$. Let F be the set of all state transformations.*

Each step of an SFC is labeled with a (possibly empty) set of *action blocks,* which are pairs of action names and qualifiers. We associate either a unique state transformation or a call of a nested SFC with an action name. Moreover, we simply say *actions* when in fact referring to action names.

Definition 4.3 (Action name, action block)
An action name *is an identifier for either a state transformation $f \in F$ or an SFC $\mathcal{S}$ (defined below). An* action block *is a pair (a, q) consisting of an action name a and qualifier q, where $q \in \{\texttt{N}, \texttt{S}, \texttt{R}, \texttt{P0}, \texttt{P1}\}$.*

Let B denote the set of all action blocks. For any action block $b \in B$ we denote by b_a the projection to the action name and by b_q the projection to the qualifier.

We define a *guard* as Boolean expression reasoning about variables and step activities. Syntactically, the activity of a step s_i is tested by the expression s_i.X.

Definition 4.4 (Guard)
A guard g *is a Boolean expression reasoning about program variable states and step activities. We denote the set of all guards by* G.

For the formal syntax for sequential function charts we further have to incorporate the key points of orders on the actions as well as on the transitions. These allow to determine in which order actions are executed and which transitions will be taken if several are in conflict with each other. In conflict means, more than one alternative transition is enabled. Hence, the order on transitions rules out any non-determinism, which is, as aforementioned, not natural for PLCs. Let *SFC* be the set of sequential function charts, which are defined next.

Definition 4.5 (SFC)
A sequential function chart *(SFC) is a 7-tuple* $\mathcal{S} = (S, A, s_0, T, block, \sqsubset, \prec)$, *where*

- S *is a finite set of* steps,
- A *is a finite set of* actions, *which might be SFCs,*
- s_0 *is an* initial step *in* S,
- $T \subseteq (2^S \setminus \{\emptyset\}) \times G \times (2^S \setminus \{\emptyset\})$ *is a finite set of* transitions,
- $block : S \to 2^B$ *is an* action labeling function *which assigns a set of action blocks to each step,*
- $\sqsubset \subseteq A \times A$ *is an irreflexive total order on actions, used to define the order in which the active actions are to be executed, and*
- $\prec \subseteq T \times T$ *is an irreflexive partial order on transitions, to determine priorities on conflicting transitions.*

We uniquely represent a transition by its set of source steps, its guard and its set of target steps.

When putting a sequential function chart into formal syntax the main task is to determine the orders $\sqsubset$ and $\prec$. As mentioned in Section 4.3.1 the orders are in general implementation dependent and should be chosen in a meaningful way or according to a existing tools.

4.3.4 Semantics of SFCs with Actions

In this section we provide an operational semantics for SFCs. Let $\mathcal{S} = (S, A, s_0, T, block, \sqsubset, \prec)$ be an SFC, and let $\mathcal{S}_i = (S_i, A_i, s_{0,i}, T_i, block_i, \sqsubset_i, \prec_i)$, $i = 1, \ldots, n$, be the SFCs nested recursively inside the steps of $\mathcal{S}$. For

a global, flat access to the nested structure we define the global set of steps $\bar{S} = S \cup S_1 \cup \ldots \cup S_n$ and the global set of actions $\bar{A} = A \cup A_1 \cup \ldots \cup A_n$. In the same way we extend the other components.

In the following we assume that the SFCs are nested in such a way that there do not exist any algebraic loops, e. g., there are no conflicting circular action qualifiers for action names.

There are several things we have to keep track of when observing executions of an SFC $\mathcal{S}$. First, we need information about the current state of $\mathcal{S}$, i. e., the values of its variables. Moreover, we have to know in which steps of $\mathcal{S}$ and its sub-SFCs $\mathcal{S}_1, \ldots, \mathcal{S}_n$ control resides and which actions are to be performed in a cycle.

Let us introduce some notions. It is crucial to distinguish between *ready steps* and *active steps.* Active steps are the ones control resides in and their actions will be performed. On the other hand, there might be steps where control resides in, but their actions will not be performed. The reason is, these steps belong to nested SFCs which are currently not activated, i. e., there is no action active which points to this SFC. Control is "waiting" there to resume. We call all steps where control resides in *ready steps.* Hence, each active step is also a ready step, but the converse does of course not hold.

Moreover, *active actions* are actions which will potentially be executed in the current SFC cycle. This means, unless there is no matching reset action these actions will be performed. *Stored actions* are the ones which have been tagged by an `S` qualifier and potentially keep on being active outside their step of activation.

We store the information about the state and the other information of $\mathcal{S}$ in a *configuration:*

Definition 4.6 (Configuration)
A configuration *c of $\mathcal{S}$ is a 5-tuple $(\sigma, readyS, activeS, activeA, storedA)$, where $\sigma \in \Sigma$ is the state of the variables, $readyS \subseteq \bar{S}$ is the set of ready steps, $activeS \subseteq \bar{S}$ is the set of active steps, $activeA \subseteq \bar{A}$ is the set of active actions, and $storedA \subseteq \bar{A}$ is the set of stored actions, i. e., the ones which might remain active outside the step they were called. Moreover, let C denote the set of all configurations.*

Such a configuration is modified in the *cycles* of a PLC. In a cycle the following sequence is performed:

1. Get new input from the environment and store the information into the state of the variables σ.
2. Execute the set $activeA$ of active actions and update σ accordingly.
3. Determine the set of next $readyS$, $activeS$, $activeA$, and $storedA$.
4. Send the outputs to the environment by extracting the required information from the new state σ.

We do not specify formally how to interact with the environment, but focus on items 2 and 3.

First, we define *executions* by means of configuration changes within a cycle. In order to do so, we associate a transition system to an SFC and use the following notations: For every transition of the form (χ, g, χ') where χ and χ' are sets of steps let $source(\chi, g, \chi') = \chi$ denote the set of all source steps and $target(\chi, g, \chi') = \chi'$ the set of all target steps. We extend these notions to sets of transition in the natural way.

A prerequisite for a program evaluation is an interpretation of the guards. We interpret a guard by the set of configurations it describes.

Definition 4.7 (Guard interpretation)
Given a guard $g \in G$ and a configuration $c \in C$, we say that "c satisfies *g" or "g* is valid in *c", denoted by $c \models g$, if $c \in g$.*

The semantics of an SFC is based on the transition system describing its operational behavior:

Definition 4.8 (Transition System of an SFC)
With every SFC $\mathcal{S} = (S, A, s_0, T, block, \sqsubset, \prec)$ we associate a transition system $\mathcal{E}(\mathcal{S}) = (C, c_0, \longrightarrow)$, where C is the set of configurations, $c_0 \in C$ is the initial configuration and $\longrightarrow \subseteq C \times C$ is the transition relation such that the following properties are satisfied: $(\sigma, readyS, activeS, activeA, storedA) \longrightarrow (\sigma', readyS', activeS', activeA', storedA')$ iff

1. *$\sigma' = (a_m \circ \cdots \circ a_1)(\sigma)$, where $a_1 \sqsubset \cdots \sqsubset a_m$ and $\{a_1, \ldots, a_m\} = activeA \setminus SFC$*
2. *$readyS' = (readyS \setminus source(taken)) \cup target(taken)$, where*
   (a) *$enabled = \{(\chi, g, \chi') \in \bar{T} \mid \chi \subseteq activeS \wedge (\sigma', readyS, activeS, activeA, storedA) \models g\}$, and*
   (b) *$taken = \{t = (\chi, g, \chi') \in enabled \mid \neg\exists t_1 = (\chi_1, g_1, \chi_1') \in enabled : \chi \cap \chi_1 \neq \emptyset \wedge t_1 \bar{\prec} t\}$ and*
3. *$activeS'$, $activeA'$ and $storedA'$ are computed by*

$$\begin{aligned}(\alpha_0, \beta_0) &:= \varphi(\alpha_{ini}, \emptyset, \mathcal{S}) \\ (\alpha_1, \beta_1) &:= \varphi(\alpha_0, \beta_0, \mathcal{S}_1) \\ &\vdots \\ (\alpha_k, \beta_k) &:= \varphi(\alpha_{k-1}, \beta_{k-1}, \mathcal{S}_k)\end{aligned}$$

where φ is defined in Fig. 4.6, $\mathcal{S}$ is the top-level SFC, $\{\mathcal{S}_1, \ldots, \mathcal{S}_k\} = storedA \cap SFC$ are the SFCs in $storedA$, and α_{ini} is the initial mapping of stored actions $a \in storedA \setminus SFC$ to $\{\mathcal{S}\}$ and to $\emptyset$ for all other actions.

Then, $activeS' = \beta_k$, $activeA' = \{b_a \mid \alpha_k(b_a) \cap \{\texttt{N}, \texttt{P0}, \texttt{P1}\} \neq \emptyset \wedge \texttt{R} \notin \alpha_k(b_a)\}$ *and* $storedA' = \{b_a \mid \texttt{S} \in \alpha_k(b_a) \wedge \texttt{R} \notin \alpha_k(b_a)\}$.

Moreover, the initial configuration $c_0 = (\sigma_0, readyS_0, activeS_0, activeA_0, storedA_0)$ *is given by:* σ_0 *is the pre-defined initial state of the variables,* $readyS_0 = \bar{s}_0$, *and* $activeS_0$, $activeA_0$ *and* $storedA_0$ *are determined from* (α_0, β_0) *like the primed ones above.*

function $\varphi(\alpha, \beta, \Gamma)$
((I) newsteps are the local ready steps, i. e., the active ones *)*
  $newsteps := S \cap readyS'$;
  $\alpha' := \alpha$;
  $\beta' := \beta \cup newsteps$;
((II) collect the relevant actions *)*
  for all $s \in S$, $b \in block(s)$:

$$\alpha'(b_a) := \alpha'(b_a) \cup \begin{cases} \{\texttt{P0}\} & \text{if } b_q = \texttt{P0} \wedge s \in source(taken) \\ \{\texttt{P1}\} & \text{if } b_q = \texttt{P1} \wedge s \in target(taken) \\ \{b_q\} & \text{if } b_q \in \{\texttt{N}, \texttt{S}, \texttt{R}\} \wedge s \in newsteps \end{cases};$$

((III) go into recursion for every new SFC which is not reset *)*
  for all $s \in newsteps$, $b \in block(s)$:
    if $b_a \in SFC \wedge \texttt{R} \notin \alpha'(b_a) \wedge \alpha'(b_a) \neq \emptyset$
    then $(\alpha', \beta') := \varphi(\alpha', \beta', b_a)$;
return (α', β')

Figure 4.6: Recursive action labeling collection

Let us take a detailed look at the definition above. One transition in the transition system above corresponds to one PLC cycle. First of all in step 1 of Definition 4.8 the current active actions are ordered and executed accordingly. In step 2 we determine the new ready steps. These are the old ones plus the targets of the taken transitions, but without their source steps. We distinguish *taken* and *enabled* transitions as follows: A transition is enabled if it is an outgoing transition of an active step and its guard satisfies the current configuration. A transition is taken, if it additionally has the highest priority among its competing ones, which is expressed by the $\bar{\prec}$ relation.

In step 3 the new active steps, active actions and stored actions are computed recursively on the structure of the SFC by the auxiliary function φ. The function φ has the parameters α, β and Γ, where α is a mapping from action names to sets of qualifiers and $\beta \subseteq \bar{S}$ is a set of steps. Calling φ for a specific SFC Γ with empty α and β results the pair (α', β'), where α' contains for every active action all the "activated" qualifiers of the active actions in Γ and its recursively nested SFCs and β' contains all active steps.

This is computed as follows: starting on with the top-level of Γ, we compute in (I) *newsteps*, which is the set of active steps on this level. The

function α is copied to α' and β' is the old β conjoined with the new active steps. In (II), α' is extended by further qualifiers for specific actions. The inclusion of qualifiers depends on whether these qualifiers become relevant now or not. The pulse qualifiers are relevant when taking a transition. The other qualifiers are relevant when remaining in the active steps (i. e., *newsteps*). Finally, in (III) we go into recursion for every sub-SFC of the current one which has become active after computing the new active steps and implicitly the new active actions. These are the SFCs associated to the new steps which are not reset.

In steps 3 of Definition 4.8 φ is called for the initial set of α, which is just the mapping of stored actions to the corresponding qualifier and the empty set for β. φ then needs to be called again for all k stored SFCs which are in *storedA*. The result of the φ application is (α_k, β_k), where β_k corresponds to the new active steps, and *activeA*$'$ and *storedA*$'$ are the projections of α_k to the corresponding qualifiers taking into account any resets.

An execution sequence of an associated transition system is called *run*.

Definition 4.9 (Run of $\mathcal{E}$)
A run *r over a transition system $\mathcal{E}$ is a finite or infinite sequence $\langle c_1, c_2, \ldots\rangle$ where each pair $(c_i, c_{i+1}) \in \longrightarrow$, i. e., is in the transition relation of $\mathcal{E}$.*

The operational semantics $[\![\mathcal{S}]\!]$ for an SFCs $\mathcal{S}$ is given by the set of runs of its associated transitions system $\mathcal{E}(\mathcal{S})$.

4.3.5 Example

For the SFC given in Fig. 4.5 we explain how the semantics works for crucial points such as actions where the execution order matters and hierarchy of SFCs.

Assume the global order on actions is given by $a_1 \sqsubset a_2 \sqsubset a_3$, and the local order on transitions is given by $g_1 \prec g_2$. Let the initial configuration for the SFC be:

- $\sigma_0 = \{x_0, y_0, z_0\} = \{0, 0, 0\}$,
- $readyS_0 = \bar{s}_0 = \{s_0, s_{10}\}$,
- $activeS_0 = \{s_0\}$,
- $activeA_0 = \{a_2\}$ and
- $storedA_0 = \{a_2\}$.

Assume, g_1 and g_2 are not true until we are in cycle number 14, but then both guards g_1 and g_2 evaluate to true. At this point

$$\sigma = \{x, y, z\} = \{13, 0, 0\},$$

the sets of active steps, ready steps etc. have not changed compared to the initial configuration. We compute according to step 1 of Definition 4.8:

$$\sigma' = (a_2)(\sigma) = \{x', y', z'\} = \{14, 0, 0\}$$

and according to step 2 of the same definition:

$$\mathit{readyS}' = (\{s_0, s_{10}\} \setminus \{s_0\}) \cup \{s_1\} = \{s_1, s_{10}\}.$$

Moreover, $\mathit{enabled} = \{g_1, g_2\}$, and since $g_1 \prec g_2$, we have $\mathit{taken} = \{g_1\}$. For the computation of the new active steps and actions we have to take the nested SFC_1 into account and recursively determine $\mathit{activeS}' = \{s_0\}$, $\mathit{activeA}'$ and $\mathit{storedA}'_0$ using the auxiliary function φ. Because $\mathit{storedA} \cap SFC = \emptyset$, i. e., there is no stored SFC, we only need to compute $\varphi(\alpha_{ini}, \emptyset, SFC_0)$ for the top level SFC which is SFC_0. The initial mapping α_{ini} of stored actions is given by

$$\begin{aligned}
&\alpha_{ini}(a_1) = \emptyset \\
&\alpha_{ini}(a_2) = \{\texttt{S}\} \\
&\alpha_{ini}(a_3) = \emptyset \\
&\alpha_{ini}(SFC_1) = \emptyset.
\end{aligned}$$

First, we determine the active steps of the top level SFC by

$$\mathit{newsteps} = \{s_1, s_2, s_3\} \cap \{s_1, s_{10}\} = \{s_1\}.$$

We set $\alpha' = \alpha_{ini}$ and $\beta' = \{s_1\}$. To update α', all relevant action qualifiers are collected. This yields

- $\alpha'(a_3) = \{\texttt{P0}\}$, since $b_q(a_1, \texttt{P0}) = \texttt{P0}$ and $s_0 \in \mathit{source}(\mathit{taken})$,
- $\alpha'(a_1) = \{\texttt{P1}\}$, since $b_q(a_1, \texttt{P1}) = \texttt{P1}$ and $s_0 \in \mathit{target}(\mathit{taken})$, and
- $\alpha'(SFC_1) = \{\texttt{N}\}$, since $b_q(SFC_1, \texttt{N}) \in \{\texttt{S,R,N}\}$ and $s_1 \in \mathit{newsteps}$.

Next, we go into recursion because there is a new SFC, which is not reset, i. e., $b_a = SFC_1 \in SFC$, $\texttt{R} \in \alpha'(SFC_1)$ and $\alpha'(SFC_1) \neq \emptyset$. Therefore, we compute $(\alpha', \beta') = \varphi(\alpha', \beta', SFC_1)$. This results in

$$\mathit{newsteps} = S_1 \cup \mathit{readyS}' = \{s_{10}\},\ \alpha' = \alpha \text{ and } \beta' = \{s_1, s_{10}\}.$$

The update of α' by collecting the relevant actions for all steps of SFC_1 yields

- $\alpha'(a_2) = \{\texttt{S,R}\}$, since $b_q(a_2, \texttt{R}) \in \{\texttt{S,R,N}\}$ and $s_{10} \in \mathit{newsteps}$ and
- $\alpha'(a_3) = \{\texttt{P0,S}\}$, since $b_q(a_3, \texttt{N}) \in \{\texttt{S,R,N}\}$ and $s_{10} \in \mathit{newsteps}$.

Since there are no new SFCs, the function terminates and returns $\beta_0 = \beta'$ as given above and $\alpha_0 = \alpha'$ with

$$\alpha_0(a_1) = \{\texttt{P1}\}$$
$$\alpha_0(a_2) = \{\texttt{S,R}\}$$
$$\alpha_0(a_3) = \{\texttt{P0,S}\}$$
$$\alpha_0(SFC_1) = \{\texttt{N}\}$$

Hence, we obtain

- $activeS' = \beta_0 = \{s_1, s_{10}\}$,
- $activeA' = \{a_1, a_3, SFC_1\}$ and
- $storedA' = \{a_3\}$.

Note, if we continue to compute the configuration for the next cycle 15, the order on actions is important as more than one action is active:

$$\sigma' = (a_3 \circ a_1)(\sigma) = \{x', y', z'\} = \{14, 0, 1\}.$$

In this case a different order on a_1 and a_3 would lead to a different σ'. Furthermore, since we do not reset a_3 which is set in step s_{10} of SFC_1 this action will still be executed even if SFC_1 is no longer active.

4.3.6 Extension to Timed SFCs

The SFCs presented earlier do not support any time or timing behavior. However, the IEC standard defines the notion of time and timers which allows to test and reason about the amount of time a step has been active and to start actions after a delay of time or for some limited time only. In this section we give an introduction to timed SFCs. We explain the main features and how to extend the current SFC semantics to timed SFCs. Since we are not going to use timed SFCs in the remainder of this work, we do not present a comprehensive formal model for timed SFCs but describe a possible extension of the untimed framework.

Timed Syntax

Guards are Boolean expressions over variables, where s_i.X denotes that step s_i is active and s_i.T is the time that s_i has been active since its last activation. Additionally to the qualifiers presented in Section 4.3.1 there are a number of time related qualifiers:

- `L` – ***L**imited*
- `D` – ***D**elayed*

- `SL` – ***S**et **L**imited*
- `SD` – ***S**et **D**elayed*
- `DS` – ***D**elayed **S**et*

All these qualifiers are followed by some duration and their meaning is as follow: The limited qualifier `L` behaves as the standard non-stored qualifier `N` with the only difference, that the corresponding action becomes at latest inactive when the associated duration has elapsed. moreover, it is possible to delay the activation of an action for a certain time after the step has become active using the `D` qualifier. The `L` and `D` qualifiers can be combined with `S` yielding `DS`, `SD` and `SL`. An action associated with `DS` will only be activated if the delay time is reached *before* the step is left, whereas an `SD` action always becomes active after the elapsed time, independent of the step activity. Similar, an `SL` action is (in contrast to `L`) *always* activated for T time units.

Although the action qualifier concept is defined using well-understood function blocks, the definitions in the standard still are ambiguous. For example, assume an SFC which calls an `SD` action in one step and in the next step calls the action again with the `SD` qualifier but with a different delay time. Assume that the activity time of the first step has been shorter than the first delay time, i.e. the action is not yet activated when it is called in the next step with a different delay time. Which delay time now is relevant? These ambiguities clearly show that there is a need for an SFC semantics which gives answers to these open questions. Here, we assume that any call of a timed action resets the corresponding clock/stopwatch. This means delays or limited actions restart the moment they got called. However, other interpretations are possible.

To reason about timed SFCs we introduce the notions of *clocks* and *stopwatches*.

Definition 4.10 (Stopwatch, Clock)
A stopwatch θ_s *is a real valued variable that has a dynamical behavior given by either* $f(\theta_s) = \dot{\theta}_s = 1$ *if the stopwatch is running, or* $f(\theta_s) = \dot{\theta}_s = 0$ *if it is stopped, whereas a* clock θ_a *cannot be stopped, i.e., the dynamical behavior is always* $f(\theta_a) = \dot{\theta}_a = 1$*. Both can be reset to zero.*

To describe the values of all stopwatches and clocks we use the notion of a *clock evaluation* ν, which is a function assigning a value to each stopwatch or clock.

Since guards may reason about the time a step is active or has been active before, each step needs a stopwatch which is reset to zero when the step is entered, runs when the step is active, and is stopped when the step is deactivated. Second, we need stopwatches because we define that hierarchically nested SFCs have history, i.e., after deactivation nested SFCs are

activated in the step the have been last. The notion of history includes that the SFCs remember the values of their clocks at the point of deactivation if they are activated again.

Additionally we need a clock for those actions which are associated with an `SD` or `SL` qualifier, because these actions might be deactivated (`SL`) or activated (`SD`) independently from a step activity. Clocks are sufficient, because according to the standard, guards cannot access the time an action has been active, and, therefore, we do not need to memorize the activity time after the action has been deactivated.

We define a timed SFC as follows:

Definition 4.11 (Timed SFC)
A sequential function chart *(SFC)* $\mathcal{S} = (X, \Theta, S, s_0, G, T, A, block, \sqsubset, \prec)$ *consists of:*

- *a finite set* X *of* variables,
- *a finite set* Θ *of* stopwatches *and* clocks,
- *a finite set* S *of* steps s_i,
- *an* initial step $s_0 \in S$,
- *a finite set* G *of* guards g_i,
- *a finite set* T *of* transitions t_i,
- *a finite set* A *of* actions a_i,
- *an* action labeling function *block assigning a set of* action blocks *to each step,*
- *an* order on actions $\sqsubset$ *to define the execution order of conflicting actions, and*
- *an* order on transitions $\prec$ *to determine priorities on conflicting transitions.*

Timed Semantics

The global state of an SFC (including all its nested SFCs) is given by the values of all its variables, the evaluation of clocks and stopwatches, the sets of active and ready steps and the sets of active and stored actions. In addition to stored actions we also have to remember *stored delayed actions* associated to a step with an `SD` qualifier, which potentially need to be activated after the corresponding step has been deactivated and *stored limited actions* indicated by the `SL` qualifier, which potentially have to be executed for a certain time after the activating step has been deactivated.

We describe the global state of an SFC $\mathcal{S}$ by a configuration, given by:

Definition 4.12 (Timed Configuration)
A configuration c *of* $\mathcal{S}$ *is an 8-tuple* $(\sigma, \nu, readyS, activeS, activeA, storedA, storedDA, storedLA)$, *where*

- σ *is the state of the variables,*
- ν *is the evaluation of clocks and stopwatches,*
- $readyS \subseteq \bar{S}$ *is the set of ready steps,*
- $activeS \subseteq \bar{S}$ *is the set of active steps,*
- $activeA \subseteq \bar{A}$ *is the set of active actions,*
- $storedA \subseteq \bar{A}$ *is the set of stored actions,*
- $storedDA \subseteq \bar{A}$ *is the set of stored delayed actions, and*
- $storedLA \subseteq \bar{A}$ *is the set of stored limited actions.*

We use the notions $\bar{S}$ and $\bar{A}$ to denote the sets of all steps or actions of an SFC including the sets of steps/actions of its nested SFCs.

The timed SFC program executions are again defined by means of configuration changes within a cycle. We describe the change of configuration by associating a transition system to an SFC. A transition $c \longrightarrow c'$ in this transition system is computed by the following sequence:

1. Determine the new state σ' by executing all actions in $activeA$ except for those which are SFCs. Conflicting actions have to be executed in accordance with the order $\sqsubset$.

2. Determine $readyS'$. To do so, determine the set $T_{enabled}$ of transitions with source steps in $activeS$ and guards holding for the current configuration. Then, $readyS'$ is given by joining $readyS$ with the sets of target steps of the transitions $t \in T_{ready}$ for which no other transition $t_1 \in T_{enabled}$ with higher priority exists, and removing the source steps of these transitions.

3. Determine $activeS'$, $activeA'$, $storedA'$, $storedDA'$ and $storedLA'$. The new sets are computed recursively on the structure of the SFC using an auxiliary function. The function recursively searches the top-level SFC and the hierarchically nested ones to find out if an action activated on a higher level is reset within an hierarchical lower active SFC.

4. Stopwatches and clocks are reset to zero, if one of the following holds:

   - The stopwatch θ_s belongs to a step s which has been activated in this cycle, i.e., $s \in activeS' \setminus activeS$.
   - The clock θ_a belongs to an action a which has been activated "stored delayed" in this cycle, i.e., $a \in storedDA' \setminus storedDA$.
   - The clock θ_a belongs to an action a which has been activated "stored limited" in this cycle, i.e., $a \in storedLA' \setminus storedLA$.

A timed execution sequence $\langle c_0 \longrightarrow c_1 \longrightarrow c_2, \ldots \rangle$ is called *timed run*. The operational semantics of an SFC $\mathcal{S}$ is given by the set of runs of its associated transition system.

4.4 Instruction List

4.4.1 Introduction

Instruction List is often considered to be the most basic language defined in IEC 1131-3, since every other programming language is expected to be mappable to IL. This is, however, not fully true. There is no way to describe parallelism as it occurs in SFCs, however, one might argue whether it is possible to simulate the sophisticated scheduling of SFC programs by IL.

Since IL is a simple assembly language, it is the language of choice to develop compact, time-critical code close to hardware. In fact, some PLCs can download IL programs directly without the intermediate compile step. A disadvantage for most people is that IL programs are hard to read as soon as they grow in size since it is more difficult to understand the control structure and the computation than in higher level languages.

IL supports a number of data types such as Booleans, integers, floating point numbers, arrays of theses types and so on. For reasons of simplicity we restrict ourselves in this work to the two most prominent types: Booleans and integers. Next to variables IL supports the use of one distinct register call *current result* (CR). Every computation takes place in the CR. E.g., first a variable value is loaded to the CR, afterwards some operations are performed on the CR and finally the current value of the CR is stored back to some variable. For this reason the CR is dynamically typed. In contrast to most other assembly languages, IL only supports exactly one distinct register.

The PLC execution mechanism for IL program is as follows: In each PLC cycle the input is read, the *whole* IL program is executed and the output is written. Note, that this contrast the SFC execution mechanism where in each cycle all active steps are evaluated and executed, but not the whole SFC program. However, IL has to be seen as integrated into SFCs. Actions (and guards) of SFC programs might be IL programs. Hence, a complete execution of an IL program within one cycle corresponds to the execution of one SFC action.

In the following we precisely define the syntax and semantics of IL programs.

4.4.2 Syntax

Next to a declaration part, instruction list programs are sequences of statements. A statement consist of an *instruction* (operator) and an operand which can either be a variable, a constant or a jump label. Additionally, programs can be augmented by comments. An example is shown below.

instruction	**operand**	**comment**
`LD`	`x`	(* loads operand value to CR *)
`JMP`	`lab1`	(* jumps to `lab1` *)

Some instructions can be augmented by *modifiers.* There are two modifiers: N and C. The N modifier changes an operation from the original to an operation with the negated argument, i.e., negated operand value while an instruction augmented by the C modifier is only executed under the condition that the CR value is *true*. Moreover, the use of brackets is allowed to force the evaluation of sub-expressions first and, hence, to avoid auxiliary variables or additional load/store operations. However, it does not add to the expressiveness of this language and we omit this feature in the following. Table 4.1 lists the most prominent IL commands we use throughout this work.

Table 4.1: List of selected IL commands

Instruction	Modifier	Operand	Description
LD	N	variable, constant	loads operand
ST	N	variable, constant	stores operand
S		variable	sets operand to *true*
R		variable	sets operand to $false$
NOT			Boolean negation
AND	N	variable, constant	Boolean AND
OR	N	variable, constant	Boolean OR
XOR	N	variable, constant	Boolean XOR
ADD		variable, constant	addition
SUB		variable, constant	subtraction
MUL		variable, constant	multiplication
DIV		variable, constant	integer division
GT		variable, constant	comparison greater than
GE		variable, constant	comparison greater equal
LT		variable, constant	comparison less than
LE		variable, constant	comparison less equal
EQ		variable, constant	comparison equal
NE		variable, constant	comparison unequal
JMP	N, C	label	jump to label
RET			return from function (block)

4.4.3 Semantics

When talking about the semantics of an IL program it is useful to describe IL programs in terms of labeled graphs representing the control structure.

We assume that every program location contains one IL statement. We denote the set of all locations of a program P by $Locs_P$. We omit the index P if it is clear from the context.

Definition 4.13 (IL graph)
An IL graph $G_P = (N, E, n_{ini}, n_{fin}, stm)$ *of an IL program* P *is a graph build from of a set of vertices* N*, a set of edges* $E \subseteq N \times N$*, a distinct initial node* $n_{ini} \in N$ *and final node* $n_{fin} \in N$ *as well as a labeling function* $stm : N \to Locs_P$ *assigning a program location, i.e., statement, to every node.*

An IL graph is *well-formed*, if the initial node is mapped to the empty statement (representing the declaration part), the edge relation corresponds to the program structure, i.e., representing the sequential program as well as jumps to labels, with correct mapping, and the final node is as well mapped to the empty statement. Moreover, from every node with a `RET` statement there is an edge to the final node n_{fin}. Subsequently, when talking about an IL graph we assume that it is well-formed. Sometimes we abbreviate $stm(n)$ by stm_n.

By Var we denote the set of all *program variables* where we tacitly assume all variables and expressions to be well typed. By the typing, some variables are marked as input or output variables or both, and we write Var_{in} and Var_{out} for the corresponding subsets of Var. Variables that are neither input nor output variables are called *local* variables. The set of all local variables is denoted by $Var_{loc} \subseteq Var$. For the current result we introduce a distinct variable $cr \notin Var$.

States and configurations

Similar to Section 4.3 we introduce the notions of states and configurations. The difference of a state or configuration in an IL program from a state or configuration of an SFC should be clear from the context. However, whenever there might be an ambiguity we refer either to *IL states* or *SFC states*; the same holds for configurations.

Definition 4.14 (IL State)
The global IL state *contains the values of all variables and is modeled as a mapping* $\Sigma : Var \cup \{cr\} \to D$*, where* D *stands for the union of all data domains.*

We assume the values in the state to be type-consistent; we use σ as typical element of Σ. Note, the CR is dynamically typed and, thus, can take on several types.

Definition 4.15 (IL configuration)
An IL configuration $\gamma : Locs \times \Sigma \times Mode$ *of a program is characterized by*

- *a location* $l \in Locs$*,*
- *a state* $\sigma \in \Sigma$*, and*
- *a configuration of type Mode, which can be either* I*,* O *or* $\mathsf{C}(ILi)$*, where* ILi *is an IL instruction.*

The mode in the configuration is used to control the various phases of the system behavior and I stands for "input", $\mathsf{C}(ILi)$ for "calculating" an "instruction", and O for "output". We define some auxiliary functions *instr* and *op*. The function *instr* maps any IL node l to the IL *instruction* of its associated statement $stm(l)$. Complementary, the function *op* maps any IL node l to the *operand* of its associated statement $stm(l)$. We assume that any operand is either a constant, a variable or a label.

Operational semantics

We define an operational semantics for IL programs based on labeled transition systems. The nodes of the transitions systems are configurations and the transitions themselves represent the i/o behavior as well as the execution of single IL statements. The transition system is labeled to distinguish between input, output and internal transitions. Each execution of an IL program is then covered by a run in this transition system. In contrast to SFCs where every transition corresponds to one PLC cycle we describe for IL programs also the internal transitions in between.

Definition 4.16 (Labeled Transition System of IL Program)
With every IL graph G_P *we associate a* labeled transition system $\mathcal{T}_P = (\Gamma, \gamma_0, \rightarrow_\xi)$*, where*

- Γ *denotes the set of configurations,*
- $\gamma_0 \in \Gamma$ *is the initial configuration and*
- $\rightarrow_\xi$ *is the transition relation between configurations.*

The initial configuration γ_0 *is given by* $(l_0, \sigma_0, \mathsf{I})$*, where the initial state* σ_0 *evaluates all Booleans to false and all integers to* 0*. The operational rules are shown in Tables 4.2–4.7 specifying the labeled transition relation* $\rightarrow_\xi$ *between system configurations.*

The labeled transitions $\rightarrow_{?\vec{v}}$ and $\rightarrow_{!\vec{v}}$ are used to mark reading the input and writing the output variables; all other transitions are unlabeled and internal. The operator $\oplus$ denotes the "exclusive or" operation.

An execution cycle starts by reading the input (cf. rule INPUT), when the system is currently in the waiting state. Afterwards the state σ is updated by assigning values to all input variable as read from the environment and

we proceed to the next mode, the computation. During the computation phase C the values of the variables or of the CR are updated according to the operations. After performing an operation control moves to the next statement. Note, that despite of jumps and the final return statement, every statement has only one successor node in the IL graph, i.e., for a node l the successor $l' \in \mathit{Succ}(l)$ is unique. Jumps are treated as (possible) branches to nodes with the label statement. Jumps have exactly two successors and we assume that only one of the successors is a label. IL programs are executed until a return statement occurs (cf. rule RET). This statement forces a function (block) to terminate and to return to its caller. Since we consider functions only, i.e., programs that terminate with a return statement, we move from C to O where the output values are written (cf. rule OUTPUT). Afterwards, the complete cycle restarts.

Table 4.2: Operational semantics: Mode switches

$$\frac{\sigma' = \sigma[\vec{x} \mapsto \vec{v}] \qquad \vec{x} = \mathit{Var}_{in}}{(l, \sigma, \mathsf{I}) \rightarrow_{?v} (l, \sigma', \mathsf{C}(\mathit{instr}(l)))}\ \textsc{Input}$$

$$\frac{\mathit{instr}(l) = \mathit{RET}}{(l, \sigma, \mathsf{C}(\mathit{instr}(l))) \rightarrow (l, \sigma, \mathsf{O})}\ \text{RET}$$

$$\frac{\vec{v} = [\![\vec{x}]\!](\sigma) \qquad \vec{x} = \mathit{Var}_{out}}{(l, \sigma, \mathsf{O}) \rightarrow_{!\vec{v}} (l, \sigma, \mathsf{I})}\ \textsc{Output}$$

Table 4.3: Operational semantics: Basics

$$\frac{instr(l) = LD \qquad \sigma' = \sigma_{[op(l) \mapsto cr]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))}\ \text{LD}$$

$$\frac{instr(l) = LDN \qquad \sigma' = \sigma_{[op(l) \mapsto \neg cr]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))}\ \text{LDN}$$

$$\frac{instr(l) = ST \qquad \sigma' = \sigma_{[cr \mapsto op(l)]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))}\ \text{ST}$$

$$\frac{instr(l) = STN \qquad \sigma' = \sigma_{[\neg cr \mapsto op(l)]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))}\ \text{STN}$$

$$\frac{instr(l) = S \qquad \sigma' = \sigma_{[true \mapsto op(l)]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))}\ \text{S}$$

$$\frac{instr(l) = R \qquad \sigma' = \sigma_{[false \mapsto op(l)]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))}\ \text{R}$$

Table 4.4: Operational semantics: Arithmetics

$$\frac{instr(l) = ADD \qquad \sigma' = \sigma_{[cr \mapsto cr + op(l)]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))}\ \text{ADD}$$

$$\frac{instr(l) = SUB \qquad \sigma' = \sigma_{[cr \mapsto cr - op(l)]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))}\ \text{SUB}$$

$$\frac{instr(l) = MUL \qquad \sigma' = \sigma_{[cr \mapsto cr * op(l)]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))}\ \text{MUL}$$

$$\frac{instr(l) = DIV \qquad \sigma' = \sigma_{[cr \mapsto cr \div op(l)]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))}\ \text{DIV}$$

Table 4.5: Operational semantics: Boolean logics

$$\frac{instr(l) = NOT \qquad \sigma' = \sigma_{[cr \mapsto \neg cr]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \text{ NOT}$$

$$\frac{instr(l) = AND \qquad \sigma' = \sigma_{[cr \mapsto cr \wedge op(l)]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \text{ AND}$$

$$\frac{instr(l) = ANDN \qquad \sigma' = \sigma_{[cr \mapsto \neg(cr \wedge op(l))]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \text{ ANDN}$$

$$\frac{instr(l) = OR \qquad \sigma' = \sigma_{[cr \mapsto cr \vee op(l)]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \text{ OR}$$

$$\frac{instr(l) = ORN \qquad \sigma' = \sigma_{[cr \mapsto \neg(cr \vee op(l))]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \text{ ORN}$$

$$\frac{instr(l) = XOR \qquad \sigma' = \sigma_{[cr \mapsto cr \oplus op(l)]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \text{ XOR}$$

$$\frac{instr(l) = XORN \qquad \sigma' = \sigma_{[cr \mapsto \neg(cr \oplus op(l))]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \text{ XORN}$$

Table 4.6: Operational semantics: Comparisons

$$\frac{instr(l) = GT \qquad \sigma' = \sigma_{[cr \mapsto cr > op(l)]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \text{ GT}$$

$$\frac{instr(l) = GE \qquad \sigma' = \sigma_{[cr \mapsto cr \geq op(l)]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \text{ GE}$$

$$\frac{instr(l) = LT \qquad \sigma' = \sigma_{[cr \mapsto cr < op(l)]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \text{ LT}$$

$$\frac{instr(l) = LE \qquad \sigma' = \sigma_{[cr \mapsto cr \leq op(l)]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \text{ LE}$$

$$\frac{instr(l) = EQ \qquad \sigma' = \sigma_{[cr \mapsto cr = op(l)]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \text{ EQ}$$

$$\frac{instr(l) = NE \qquad \sigma' = \sigma_{[cr \mapsto cr \neq op(l)]} \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \rightarrow (l', \sigma', \mathsf{C}(instr(l')))} \text{ NE}$$

Table 4.7: Operational semantics: Jumps

$$\frac{instr(l) = LABEL \qquad l' \in Succ(l)}{(l, \sigma, \mathsf{C}(instr(l))) \to (l', \sigma, \mathsf{C}(instr(l')))}\ \text{LABEL}$$

$$\frac{instr(l) = JMP \qquad l' \in Succ(l) \qquad instr(l') = LABEL}{(l, \sigma, \mathsf{C}(instr(l))) \to (l', \sigma, \mathsf{C}(instr(l')))}\ \text{JMP}$$

$$\frac{\begin{array}{c} instr(l) = JMPC \qquad l' \in Succ(l) \\ cr(\sigma) = false \qquad instr(l') \neq LABEL \end{array}}{(l, \sigma, \mathsf{C}(instr(l))) \to (l', \sigma, \mathsf{C}(instr(l')))}\ \text{JMPC}_{\text{FF}}$$

$$\frac{\begin{array}{c} instr(l) = JMPC \qquad l' \in Succ(l) \\ cr(\sigma) = true \qquad instr(l') = LABEL \end{array}}{(l, \sigma, \mathsf{C}(instr(l))) \to (l', \sigma, \mathsf{C}(instr(l')))}\ \text{JMPC}_{\text{TT}}$$

$$\frac{\begin{array}{c} instr(l) = JMPCN \qquad l' \in Succ(l) \\ cr(\sigma) = false \qquad instr(l') = LABEL \end{array}}{(l, \sigma, \mathsf{C}(instr(l))) \to (l', \sigma, \mathsf{C}(instr(l')))}\ \text{JMPCN}_{\text{TT}}$$

$$\frac{\begin{array}{c} instr(l) = JMPCN \qquad l' \in Succ(l) \\ cr(\sigma) = true \qquad instr(l') \neq LABEL \end{array}}{(l, \sigma, \mathsf{C}(instr(l))) \to (l', \sigma, \mathsf{C}(instr(l')))}\ \text{JMPCN}_{\text{FF}}$$

4.4.4 Example Program

In this section we give a small example of an IL program. The program displayed in Figure 4.7 computes the best lower approximation of a square root of an integer.

The program works as follows: In the header the variables are declared. We have input, output and local variables. First, the PLC is updating the input variables, i.e., `x`. This is not part of the program. Then, the program starts. The local variable `v` is initialized to zero. Within the loop from `start:` to `JMPC start` the variable `v` is successively increased by 1 until its square is greater or equal than the input `x`. If equal to `x` the result of the computation is `v`, if greater than `x` the result is `v-1` since we want to compute the best lower approximation. After hitting the `RET` statement the program is terminated and the output written to the environment.

This program is a simple approximation of a square root and the square root function itself is part of the IL language, but it illustrates how an IL looks like.

Below we present an example execution of the program. We assume an input `x` $= 2$. The result is 1, since $1^2 \leq 2 < 2^2$, i.e., it is a lower approximation of the square root of 2. The run of the example execution is given as a list of configuration $\gamma : \mathit{Locs} \times \Sigma \times \mathit{Mode}$. However, we omit the locations *Locs*, this should be clear from the context.

cr	x	result	v	vsqr	*Mode*
0	0	0	0	0	I
0	2	0	0	0	C(LD)
0	2	0	0	0	C(ST)
0	2	0	0	0	C(LABEL)
0	2	0	0	0	C(LD)
0	2	0	0	0	C(ADD)
1	2	0	0	0	C(ST)
1	2	0	1	0	C(MUL)
1	2	0	1	0	C(ST)
1	2	0	1	1	C(LD)
2	2	0	1	1	C(GT)
true	2	0	1	1	C(JMPC)
true	2	0	1	1	C(LABEL)
true	2	0	1	1	C(LD)
1	2	0	1	1	C(ADD)
2	2	0	1	1	C(ST)
2	2	0	2	1	C(MUL)
4	2	0	2	1	C(ST)
4	2	0	2	4	C(LD)
2	2	0	2	4	C(GT)

false	2	0	2	4	C(JMPC)
false	2	0	2	4	C(LD)
2	2	0	2	4	C(EQ)
false	2	0	2	4	C(JMPC)
false	2	0	2	4	C(LD)
2	2	0	2	4	C(SUB)
1	2	0	2	4	C(ST)
1	2	1	2	4	C(JMP)
1	2	1	2	4	C(LABEL)
1	2	1	2	4	C(RET)
1	2	1	2	4	O

In the remainder we are concerned about proving certain properties about IL programs.

```
FUNCTION PLC_PRG_SQRT
VAR_INPUT
        x     :INT;           (* the input value *)
END_VAR

VAR_OUTPUT
        result:INT;           (* the output result *)
END_VAR

VAR
        v     :INT;           (* auxiliary value *)
        vsqr  :INT;           (* auxiliary value *)
END_VAR

        LD              0
        ST              v     (* v initialized to 0  *)
start:
        LD              v
        ADD             1     (* v increased by 1     *)
        ST              v
        MUL             v     (* compute square of v *)
        ST              vsqr  (* store square        *)
        LD              x
        GT              vsqr  (* x greater than v square?    *)
        JMPC    start         (* jump to start to increase v *)
        LD              x
        EQ              vsqr  (* x equals v square?  *)
        JMPC    equal         (* jump to 'equal' for output     *)
        LD              v     (* x must be greater than v square *)
        SUB             1     (* subtract 1 from v              *)
        ST              result(* write output                  *)
        JMP             end
equal:
        LD              v
        ST              result
end:
        RET
```

Figure 4.7: Approximated square root computation

Chapter 5

Verification

5.1 Introduction

One goal of this work is to apply software verification techniques for selected PLC languages to improve the overall reliability of control processes. The previous chapter introduced a formal semantics for the languages IL and SFCs. This semantics serves as a formal basis for the subsequent verification methods applied to programs written in those two languages. Since the nature of these two languages are quite different we also propose different verification approaches to each of them.

SFCs are a high level structuring language based on transition systems. Errors on this level are most likely to be of conceptual nature: Expected mutual exclusions do not hold, certain steps are not reachable or on contrary a step is never left. To prove the correctness of these desired properties we suggest model checking as the verification technique of choice. Model checking provides an exhaustive analysis of properties on finite state models. In general, it is easy to find a finite abstraction for an SFC program. Basically, the control structure, the activity of steps and actions are of interest. In Section 5.2 we provide a translation from abstract SFCs to the input language of the CaSMV [JM01] model checker. Moreover, we give a characterization of safe SFC and define a model checking approach to identify them.

In contrast to SFCs the language IL is a low level, hardware oriented language. There are less structuring possibilities, the code involves potentially infinite data types (integers, reals) and can consist of thousands of lines of instructions. Failures of these program are often due to programming mistakes leading to run-time errors: variables exceed their allowed range, code is unreachable or leads to infinite loops, there are typing mistakes or illegal arithmetic operations. According to the nature of the language and the properties to be checked we propose a combined method of abstract interpretation and data flow analysis for verification. The advantage of our approach is that the checking is done fully automatic and does not require

and human interaction like in, e.g., model checking.

In Section 5.3 we first define an abstract simulation for IL programs. This simulation allows to run an IL program for sets of inputs on an abstract level simultaneously instead of testing single instances. This provides practical feedback to the user when testing programs for many possible inputs. Moreover, for verification we propose a combination of abstract interpretation and data flow analysis techniques to check the generic properties. While abstract interpretation helps to determine the range of program values, data flow analysis allows to relate different program information according to specific requirements.

5.2 Sequential Function Charts

This section provides methods and techniques for the verification of SFC properties. We distinguish between *structural* and *semantic properties*, although these are interrelated.

The semantic properties we consider are mostly program dependent reachability properties such as: mutual exclusion between shared variables, termination, avoidance of undesired states etc. We validate these properties by model checking SFCs. To enabled this, we present a translation of SFC program into the input language of a model checker. This translation can be performed automatically and, thus, does not require any error prone manual work. Once translated, any temporal property of the considered SFC program can be checked. This allows the verification of a rich class of semantic properties of SFCs.

By the verification of structural properties we understand checking for a correct syntax and "sensible" SFCs. Checking for a correct syntax is easy, it is a conformance testing of the syntactic rules for building SFCs as already define in the standard IEC 61131-3. However, the syntax allows many constructions which do not make sense. Therefore, we introduce the notion of a *safe SFC*, an SFC whose structure is sensible. Moreover, we present a way to determine safe SFCs by model checking.

We start with introducing an abstract model for SFC programs. This is the basis for later analysis. Afterwards, we define a translation from the abstract model to into the input language of the model checker. This is described in Section 5.2.2. A characterization of safe SFCs and their determination is presented in the subsequent Section 5.2.3.

5.2.1 An Abstract SFC Model

In this work we are mainly interested in the SFC program organization, i.e., the steps, transitions, actions and their qualifiers. We are not concerned about the programs others than SFCs referred by the individual actions. Thus, we abstract from any concrete program, but analyze the program

activation, i.e., action activations. In the same way we abstract from guards reasoning about anything else than step activities and Boolean variables, either program or input variables. This results into a finite SFC model. We define it as follows:

Definition 5.1 (Abstract SFC)
An abstract sequential function chart *(SFC) is a 6-tuple* $\mathcal{S}^{\#} = (S, A^{\#}, s_0, T^{\#}, \mathit{block}, \prec^{\#})$*, where*

- S *is a finite set of* steps,
- $A^{\#}$ *is a finite set of* abstract actions *which might refer to SFCs and are essentially Boolean variables,*
- s_0 *is the* initial step *in* S*,*
- $T \subseteq (2^S \setminus \{\emptyset\}) \times G^{\#} \times (2^S \setminus \{\emptyset\})$ *is a finite set of* abstract transitions, *where guards are Boolean expressions over step activities and Boolean variables,*
- $\mathit{block} : S \to 2^B$ *is an* action labeling function *which assigns a set of action blocks to each step,*
- $\prec^{\#} \subseteq T \times T$ *an order on abstract transitions.*

The abstract guards $g \in G^{\#}$ are Boolean expressions over step activities and Boolean variables. Here, any concrete guard such as for instance $x > 1$ is replaced by a Boolean variable *x_gt_1*. This Boolean variable is allowed to behave arbitrarily during the program execution, i.e., it abstracts any concrete behavior. Moreover, any abstract action $a^{\#} \in A^{\#}$ is the replacement of the original action, i.e., a state transformation, by a Boolean variable indicating that can be either active or not. This means, we abstract completely from its program code.

In a first approach we also abstract from the order on transitions, i.e., we consider abstract SFCs of the form $(S, A^{\#}, s_0, T^{\#}, \mathit{block})$. This models an SFCs whose execution is independent of any priorities and allows more program behavior than there actually is on the concrete level. However, if the program satisfies some safety property on the abstract level, it certainly will do so on the concrete level. In the latter we show how to include priorities on the abstract level as well.

The *abstract semantics* of a program is given by the operational semantics as given for its concrete counterpart in Definition 4.8, but we do not consider any effects of the actions and, thus, omit, e.g., item number 1 of the definition.

Example 5.1 Consider again the SFC of Figure 4.4. The abstract counterpart of the SFC depicted there is shown in Figure 5.1. Here, the concrete guards $y = 0$, $y = 1$ and $x \leq 1$ are replaced by the Boolean variables *y_eq_0*, *y_eq_1* and *x_leq_1*. Moreover, we abstract from the the effect of actions if they do anything else than calling sub-SFCs.

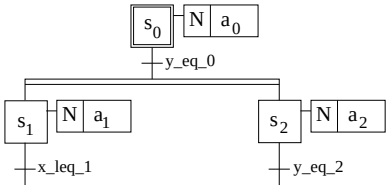

Figure 5.1: Abstract SFC

5.2.2 Translation to CaSMV

CaSMV is a symbolic model-checker which supports the verification of temporal logic properties of Kripke structures. The transition relation of a Kripke structure is expressed in CaSMV by evaluation rules depending on the current and the next state of each system variable q, i.e., `q` and `next(q)` in CaSMV notation. In order to translate an SFC to CaSMV we mimic the transition relation on the configurations of the SFC semantics. Remember, after reading the inputs, at every PLC cycle we first execute the actions and then evaluate the transitions and determine the new active steps and actions. Hence, whenever reasoning about guards or the new state of steps and actions we refer to the `next` state of a variable.

In the beginning we do not consider any order on actions and transitions. We assume that the activity of an action directly corresponds to an output variable, which is often the case as many actions only consist of opening and closing valves, switching motors on and off etc. In this case we do not need an order on actions, because the actions do not share variables. Furthermore, we first have no order of transitions which allows us to additionally check for conflicting transitions automatically. In Section 5.2.2 we show how to extend this framework by embedding orders on transitions and actions, which results into a deterministic execution model. This enables us to deal with more complex actions and situations where one variable is modified by more than one action and the result depends on the execution order.

Data Structure of the CaSMV Module

A system modeled in CaSMV can be composed from components called *modules.* We use one module to describe the SFC and its actions and allow further modules to describe the environment or parts of the environment.

In order to translate an abstract SFC $\mathcal{S}^{\#} = (S, A^{\#}, s_0, T^{\#}, \mathit{block}, \prec^{\#})$ to CaSMV we introduce the following Boolean variables:

- `ready_s`i for each step s_i, i.e., one variable for each step of the top level SFC and its hierarchically nested ones. These variables model whether the respective step is ready, this means, the step is active or control resides in it and waits to resume.

- `guard`i for each guard g_i. This variable represents the transition condition and is in general a Boolean expression reasoning about program variables and *input variables* `input`i (e.g., process variables from the plant to be controlled) and the *activity of steps* `step.X`i. Where, e.g., `step.X`*1* evaluates to true whenever step one is active.

- `active_a`i for each action a_i. This variable is introduced to code whether an action is active or not. This action might be an SFC itself.

- `stored_a`i for each action a_i, which indicates whether an action is currently stored, i.e., has been activated in the current or a previous step by an `S` qualifier.

Note, if we want to reason about the activity of a step s_i belonging to a nested SFC a_k, the variable `step.X`i will statically be substituted by `ready_s`$i \wedge$ `next`(`active_a`k). This means, a step s_i is active, if it is currently ready and after the execution of all actions the SFC it is nested in becomes (or remains) active.

Furthermore the CaSMV module has *input parameters* for each Boolean input variable of the SFC program. The behavior of the input variables is a priori chaotic, i.e., they might take any possible value, unless specified otherwise. This allows to check an open system. Any restrictions on the behavior of input variables can be modeled in an additional CaSMV module representing the environment.

Evolution of State Variables

In this section we define how to code the transition relation on the variables defined in the previous section. This is of special interest for the activity of actions, which are tagged by qualifiers. Therefore, we explicitly define the `next`-state of all variables, but guards and input variables, since inputs are provided by some environment and the truth values of guards are determined by the evaluation of the Boolean expressions they represent.

Ready steps. The `next`-state of a ready variable `ready_s`i of a step s_i is true if and only if in the `next`-state there is a transition taken into s_i or it is already true now and there is no transition leaving s_i. In case s_i belongs to a nested SFC, it is additionally required that in the `next`-state the the nested SFC itself is active. In detail, for a nested SFC associated by an action a_k the variables `ready_s`i for each step $s_i \in a_k$, are set to true only the SFC itself is active, i.e. `next`(`active_a`k) holds, one of the preceding guards will be true in the next cycle and the corresponding source steps are currently ready.

$$\begin{aligned} \texttt{next(ready\_s}i\texttt{)} &= \texttt{s}i\texttt{\_will\_be\_entered} && (5.1)\\ &\vee\ (\texttt{ready\_s}i \wedge \texttt{s}i\texttt{\_will\_not\_be\_left}) && (5.2) \end{aligned}$$

Condition 5.1 states that step s_i becomes active if it will be entered in the next cycle and 5.2 that it remains active if it is active now and no other transition is taken. We define these conditions in detail:

$$\begin{aligned} \texttt{s}i\texttt{\_will\_be\_entered} &= (\exists t = (A, g, A') \in T\ :\ s_i \in A' \\ &\qquad \bigwedge_{s_j \in A} \texttt{ready\_s}j \wedge \texttt{next(g)}) \\ &\wedge\ (\neg\exists t' = (B, g', B') \in T\ :\ t \neq t' \wedge A = B \wedge \\ &\qquad \bigvee_{s_l \neq s_i \in B'} \texttt{next(ready\_s}l\texttt{)}) \end{aligned}$$

This means, step s_i will be entered if it is in the target of a transition that is enabled in the next cycle, and there is no other transition from the same source that will be taken instead. The second conjunction of the formula above is necessary to cope, e.g., with alternative branches where several guards are true at the same time. The details of condition 5.2 are:

$$\begin{aligned} \texttt{s}i\texttt{\_will\_not\_be\_left} &= \neg\exists t = (A, g, A') \in T\ :\ s_i \in A \wedge \\ &\qquad \bigwedge_{s_j \in A} \texttt{ready\_s}j \wedge \texttt{next(g)} \end{aligned}$$

This means, step s_i will remain active if there is no outgoing transition enabled. Note, however, that 5.1 and 5.2 are defined with respect to safe SFCs as defined later.

As previously stated, if step s_i belongs to a sub-SFC a_k we have to add $\texttt{next(active\_a}k\texttt{)}$ in conjunction to 5.1 and 5.2.

Active actions. The value of $\texttt{active\_a}k$ for the activity of an action a_k depends on the activity of the steps s_j, where $a_k \in block_a(s_j)$, and the qualifiers tagged to a_k. The expression for determining $\texttt{next(active\_a}k\texttt{)}$ is defined by

$$(\texttt{act\_N\_S\_steps} \vee \texttt{act\_P1\_steps} \vee \texttt{act\_P0\_steps} \vee \texttt{stored\_a}k) \wedge \neg\texttt{act\_R\_steps}$$

where

- $\texttt{act\_N\_S\_steps}$ is $\bigvee_{\{s_j \mid a_k \in block_a(s_j)\}} (\texttt{ready\_s}j \wedge \texttt{next(active\_a}l\texttt{)})$ where a_k is tagged with the `N` or `S` qualifier and a_l is the SFC s_j belongs to,

- act_P1_steps is $\bigvee_{\{s_j \mid a_k \in block_a(s_j)\}}(\neg \texttt{ready\_s}j \wedge \texttt{next}(\texttt{ready\_s}j))$ where a_k is tagged with the P1 qualifier,
- act_P0_steps is $\bigvee_{\{s_j \mid a_k \in block_a(s_j)\}}(\texttt{ready\_s}j \wedge \texttt{next}(\neg \texttt{ready\_s}j))$ where a_k is tagged with the P0 qualifier, and
- act_R_steps is $\bigvee_{\{s_j \mid a_k \in block_a(s_j)\}}(\texttt{ready\_s}j \wedge \texttt{next}(\texttt{active\_a}l))$ where a_k is tagged with the R qualifier and a_l is the SFC s_j belongs to.

This means, an action will become active if one of the following conditions hold: The step the action is associated to will become active and the action itself is tagged with either the N or the S qualifier, if a step the action belongs to will be entered in the next cycle and the action is tagged with the qualifier P1, the step the action belongs to is active and will be inactive in the next cycle and the action is tagged with the qualifier P0, or the action is a stored one (see below). However, resetting an action has always priority and, thus, will in any case disable the activation.

Stored actions. The value stored_ak is set to true, if one or more steps where a_k is associated to are active and a_k is tagged with an S qualifier and there is no matching reset. It is set to false, whenever a matching reset action is called. Thus the next value of stored_ak is defined by $\texttt{next}(\texttt{stored\_a}k) = (\texttt{act\_S\_steps} \vee \texttt{stored\_a}k) \wedge \neg \texttt{act\_R\_steps}$ where

- (act_S_steps is $\bigvee_{\{s_j \mid a_k \in block_a(s_j)\}}(\texttt{ready\_s}j \wedge \texttt{next}(\texttt{active\_a}l))$ where a_k is tagged with the S qualifier and a_l is the SFC s_j belongs to and
- act_R_steps as defined above.

Initialization The *initial ready step* s_0 of the top-level SFC is initialized to true, denoting that this step is active at the beginning. All other steps are initially set to false. For reasons of simplicity we assume, that the initial step of the top level SFC contains no nested SFCs. This does not limit the set of translatable SFCs, because each SFC can be transformed into one meeting this constraint. Furthermore, all variables coding if the action is active or the action is a stored one are initially false.

Extension to Orders on Actions and Transitions

The translation presented above does not take into account the orders on actions, $\sqsubset$, and on transitions, $\prec$. Furthermore it only works for actions whose activity is mapped to an output variable. However, this approach can be extended to consider existing orders on transitions and actions. To take the order on transitions into account we modify the guards of the transitions such that there are no more conflicts. This can be done statically by adding constraints such that a transition is enabled iff its guard holds and no other

higher-priority transition which shares at least one common source steps is enabled.

To consider more complex actions which make it necessary to deal with the order on actions we introduce a *micro-cycle.* The micro-cycle length equals the number of actions which have to put into an order. Then, each action is triggered by the cycle according to its position in the order.

Example Application to a Chemical Plant

In this section we apply the presented approach to a laboratory plant which was designed and built at the Process Control Laboratory of the University of Dortmund to serve as a test-bed and a demonstration medium for control and scheduling methods for multi product batch plants [BKSL00]. Compared to batch plants of industrial scale this plant is still of moderate size although it is already complicated enough to pose complex scheduling and control tasks.

In the plant two products are produced from three raw materials in three reactors simultaneously. The plant itself may be seen as part of a production line, therefore, there are buffer tanks for the three raw materials which are assumed to come from a preceding step of this production line. Additionally there are buffer tanks for the products which are supposed to be consumed in a following production step.

For illustration purposes we only focus on one part of the plant, the production of one product in one of the reactors.

Example Process and Control Program In Fig. 5.2 the reactor T3 is depicted, which is used to produce the chosen product, referred to as C, from two raw materials, referred to as A and B. The tanks T1 and T2 are used as buffer tanks for A and B respectively. They can be filled via the valves V1 and V2. The reaction is carried out by first filling A into T3 via valve V3. After this the contents of tank T2 is given into the reactor via V4, now B immediately reacts with A to C. Product C then directly can be withdrawn via V5 for further processing. Tanks T1 and T2 are equipped with sensors LIS+ for detecting the upper liquid level threshold and LIS- for the detecting that the tank is empty. Apart from an LIS- sensor tank T3 is also equipped with a stirrer M, which is used to ensure a homogeneous solution within tank T3 during the reaction.

In Fig. 5.3 a control program for carrying out the reaction is shown. The control program consists of a kind of master-SFC, which allows the following three processes to run in parallel

- *filling T1 with A* given by action a_1 in step s_2,
- *filling T2 with B* given by action a_2 in step s_5 and

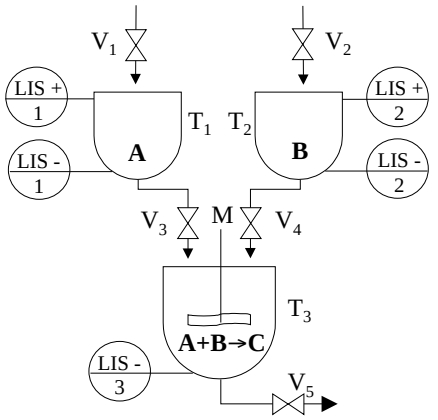

Figure 5.2: The plant

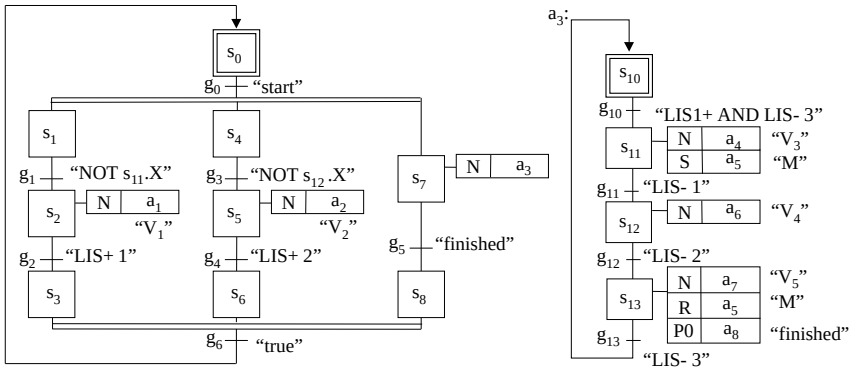

Figure 5.3: Control SFCs

- *reaction in T3 and emptying T3* given in step s_7 as action a_3.

Action a_3 is given by an individual SFC, since this process itself has a certain complexity. As we have conflicting processes, such as *emptying contents of T1 into T3*, which is a sub step of a_3 and *filling T1 with A* we have the waiting steps s_1, s_4 and s_{10} which shall ensure that certain conditions (given as guards) hold before the processes start.

Apart from a_3 the actions are very simple. For these the value of actions activity simply determines the value of an output variable, e.g. the valve V1 is as long open, as a_1 is active.

Note, the control routines presented differ from those given in [BKSL00] as they are adapted to the excerpt of the plant presented.

Translation The translation of the control program into CaSMV code follows directly from the definitions of Sect. 5.2.2. Here, we give just two examples of how to define the transition relation on state variables. The

CaSMV code for these examples are at Fig. 5.4, where the symbols '`&`', '`|`', '`~`' (as well as '`!`') denote logical 'and', 'or' and 'not'. For the whole example the code can be found at the Appendix A.

Steps. Step s_{12} of the nested SFC will become ready, if the preceding step s_{11} is currently ready, the SFC it is nested in will be active and the guard *LISminus1* of the transition connecting these two steps will evaluate to true. On the other hand step s_{12} will become inactive, if it is currently ready, its SFC will be active and the outgoing transition condition will hold. If none of these cases is satisfied s_{12} will keep its current status.

Actions. Action a_5 will become active if either s_{11} is active or a_5 is already stored and s_{13} is not a ready step, since a_5 is reset in s_{13}. In any other case s_{13} will be inactive.

```
default next(ready_s12) := ready_s12;
 in case{
    (ready_s11 & next(active_a3)
    & next(LISminus1)) : next(ready_s12) := 1;
    (ready_s12 & next(active_a3)
    & next(LISminus2)) : next(ready_s12) := 0; }


default next(active_a5) := 0;
 in case{
    (((ready_s11 & next(active_a3)) | stored_a5)
          & ~ready_s13) : next(active_a5) := 1; }
```

Figure 5.4: Fraction of CaSMV input

Specification of Verification Tasks For the given SFCs we check the following properties:

Reachability of each step. We check this to ensure that there is no unused code. The corresponding CTL specification added to the CaSMV input file is for a step s_i: `SPEC EF si`, i.e., on all execution paths we will eventually reach s_i.

Absence of deadlocks. We check that whenever a step s_i is reached it is possible to extend this run such that s_i is reached once more, i.e., infinitely often. In CaSMV this is specified by: `SPEC AG (AF si)`.

Plant specific requirements. For the design of control programs of batch plants the allocation of plant equipment to different production steps or processes is a central issue. There are often processes, which are in conflict because they need the same resources. E.g., *emptying contents of T1 into T3* and *filling T1 with A* are in conflict, because they both compete for tank T1. Therefore, it has to be checked that equipment is exclusively used by one process at a time. In the given example it has to be ensured that both valves for filling and emptying a tank are not open simultaneously. E.g., for tank T1 the valves V1 and V3 shall never be open at the same time, which leads to the specification: `SPEC AG !(V1 & V3)`.

The verification tasks presented here are independent of a specific environment but reason about the control software only. In order to verify, e.g., that there is no overflow in a tank, parts of the plant and the environment have to be included into the model and checked in combination with the current controller.

Verification Results All verification tasks presented above are checked within a fraction of one second on a SUN ULTRA 1. This is not surprising, since the model is still of small size and for illustration purpose only.

It was proven that every step is reachable and there are no deadlocks. We also verified that the tanks T1 and T3 are never filled and emptied at the same time. However, tank T2 does not satisfy this requirement. The counter trace produced by CaSMV shows that both valves V2 and V4 may be opened simultaneously. This happens, because it is only required when entering step s_5 that step s_{12} is not active (`NOT` s_{12}`.X`), i.e. that filling T2 does not start, if it is already in the emptying phase. However, when entering s_{12} there is no condition that checks, if the tank is still in the filling phase.

Hence, the verification detects a flaw in the control program which is not obvious to see and the counter trace helps to track back its origin, where it can be remedied.

5.2.3 Safe Sequential Function Charts

In Figure 4.3 of Section 4.3 we first presented an SFC whose structure complies to the syntax as given in IEC 61131-3, but which does not make sense. We like to characterize what kind of constructions do not make any sense. Any SFC without a divergence/convergence is always sensible as long as the syntactic rules are followed. However, if we allow parallelism there are three major violation that might occur:

1. There is a jump between different parallel branches without a proper synchronization first. This violates a proper parallelism and does not

make sense (cf. Figure 5.5 SFC a).

2. There is a jump out of a parallel branch, probably to some other level. This violates a proper synchronization (cf. Figure 5.5 SFC b).

3. Two or more alternative branches of the same parallel branch are synchronized. This does not make sense either (cf. Figure 5.5 SFC c).

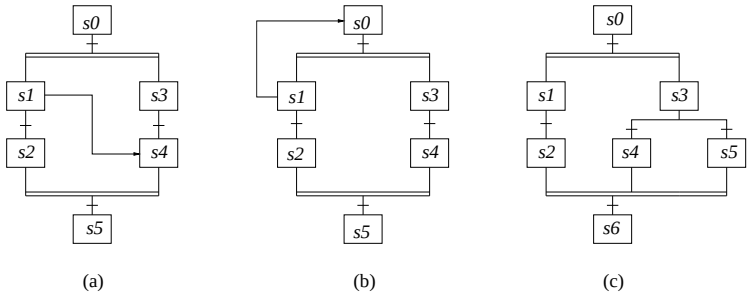

Figure 5.5: Various types of unsafe SFCs

This leads to the following characterization:

Characterization 5.1 (safe SFC)
An SFC is safe *if there are no jumps between parallel branches, no jumps out of parallel branches and every branch is properly synchronized.*

Although we have characterized safe SFCs we have not yet presented a way to automatically determine if an SFC is safe and what are the semantic consequences of the characterization above. This is done in the following.

A Model Checking Approach to Safe SFCs

The first two items of the characterization above describe a breach of proper parallelism. If we consider parallel branches as independent *processes* we clearly want control not to jump from one process to the other or to abort when in fact we demand a proper synchronization.

Semantically, this means that control might be simultaneously at different places in the same process. To understand this, imagine the following scenario in Figure 5.5 (a): Control starts from step $s0$ moves on to $s1$ and $s3$. While in the right branch control remains in $s3$ it moves from the left in $s1$ to the right in $s4$. Now, control resides in two steps in the same process, i.e., in $s3$ and $s4$. In the middle SFC we can construct a similar scenario with the same result.

If we consider control as being *tokens* moved around according to the SFC structure, we can phrase one verification condition for a safe SFC as follows: There is never more than one token simultaneously in the same

process. If we completely abstract an SFC from guards and actions and allow control to reside in a location as many cycles as wished, we can rephrase the verification condition as: Determine that never more than one token is in the same step. This abstraction to SFCs without guards where control might reside in a step arbitrarily long is a sensible abstraction, since we want to verify structural properties that do not depend on specific guards and actions.

The issue of proper synchronization is slightly different. The SFCc in Figure 5.5 is not unsafe because control might reside simultaneously in two processes, but since there are not enough tokens for the convergence, which is of course due to the alternative branches. Hence, a verification condition for this problem is that there is a possibility to reach simultaneously all source steps of a converging transition. Again it is useful to consider SFCs abstracted from any guards and actions.

Definition 5.2 (safe SFC)
An SFC is safe *if for all possible runs there is at most one token in a process and for all converging transitions there exists a run such that they can be taken.*

Both verification conditions are stated as reachability problems over finite graphs and, hence, are solvable by model checking. Since we already defined a translation from SFCs to the input language of the CaSMV model checker we are going to make use of this and modify the translation slightly to check for safe SFCs.

A modification of the translation. As mentioned, we check the structure independent of any guards, i.e., control might move in an arbitrary way. Therefore, we substitute in the translation process every guard by *true*. However, this alone is not sufficient. By the synchronous execution mechanism in every cycle every enabled transition is taken. This means in particular, if all guard evaluate to *true* control moves in every cycle out off a step (if possible). However, we like the control to behave arbitrarily, in particular it might remain in a step. Therefore, we introduce a self-loop guarded by *true* for every step. This allows control either to stay in a step, i.e., to self-loop, or to take an outgoing transition.

Token overflow flag. In preparation to check that there might be more than one token in a step at a time we introduce an additional variable `token_overflow` to indicate just this. We define `token_overflow` such that it is set to 1 (*true*) whenever a step $s_i \in S$ is ready and there is a transition t leading to s_i and the source steps of t are ready steps themselves. Once set to 1 `token_overflow` remains 1. It is initialized by 0. Formally, the transition

relation is defined on all steps s_i which have an incoming transition by:

$$\begin{aligned} \texttt{token\_overflow}' \;=\; & \bigvee_{s_i \in S} \texttt{ready\_s}_\texttt{j}\Big(\bigvee_{\{\texttt{t}=(\texttt{A},\texttt{g},\texttt{A}') \in \texttt{T} \mid \texttt{s}_\texttt{i} \in \texttt{A}'\}} \bigwedge_{\texttt{s}_\texttt{j} \in \texttt{A}} \texttt{ready\_s}_\texttt{j} \Big) \\ \vee\; & \texttt{token\_overflow} \end{aligned}$$

This means we have a possible token overflow in s_i, whenever s_i is ready and so are the target steps of a transition leading to s_i. The reason is, in the next cycle the token from the target step might move to s_i, while the token in s_i might remain in s_i due to the self-loop. This results into more than one token in s_i.

Verification conditions. To check for token overflow and proper synchronization we define temporal logic properties on the new extended transition system. Clearly, there are not more than one token in a step at a time if there is always no token overflow, i.e.:

```
SPEC AG !token_overflow.
```

The verification property for proper synchronization is slightly more complex. We have to check for every converging transition $t = (A, g, A') \in T$ with $|A| > 1$ that all steps in A might ready. Let the k-element set of all converging transitions be

$$T_k = \{t \mid t = (A, g, A') \in T \wedge |A| > 1\}$$

such that $|T_k| = k$. We require:

$$\texttt{SPEC } |_{t=(A,g,A') \in T_k} \texttt{EF } \&_{s_i \in A} \texttt{ready\_s}_i \,.$$

This means for all converging transition eventually there are all source steps ready. Note, however, this need not to be the case for the concrete SFC. Here, we abstract from any restrictions imposed by guards etc. and, therefore, over-approximate the behavior to test for the structural compliance.

Example. Let us consider SFC a of Figure 5.5. The translation relation for a token overflow as well as the verification conditions are shown in Figure 5.6

The verification result is, of course, that there is proper termination but also that there is a possibility of a token overflow. When taking a look at SFC c of Figure 5.5 on might suggest that detecting proper synchronization equals detecting a token overflow in the reversed transition relation. This means, we change the direction of all transitions and then check for a token overflow. Although this works for the examples in Figure 5.5 this is generally not possible. A counter example is shown in Figure 5.7.

```
default next(token_overflow) := token_overflow;
 in case{
    (ready_s1 & ready_s0 :=1;
    (ready_s2 & ready_s1 :=1;
    (ready_s3 & ready_s0 :=1;
    (ready_s4 & (ready_s3 | ready_s1) :=1;
    (ready_s5 & (ready_s2 & ready_s4) :=1;}

(* SPEC token overflow *)
SPEC AG ! token_overflow

(* SPEC proper synchronization *)
SPEC EF (ready_s2 & ready_s4)
```

Figure 5.6: SMV code for token overflow

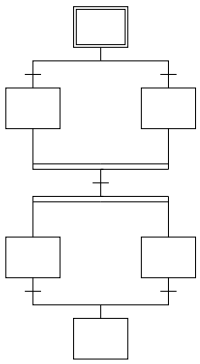

Figure 5.7: Completely unsafe SFC

5.3 Instruction List

Run-time errors are assumed as a particularly high-risk type of software fault whose consequences include processor halt, data corruption and security breaches. They also cause applications to send uncontrolled commands to external devices, causing non-deterministic, unpredictable behavior. Impacts on business and corporate image may be catastrophic (due to loss of service, loss of mission, etc.) [Pol02].

In this section we present a number of ways to detect possible run-time errors of IL programs. To test IL programs we start with the development of an abstract simulation framework in Section 5.3.1. This framework is extended to abstract interpretation in Section 5.3.2 which allows to automatically check for a number of properties such as range violation or division by zero. A heuristics to improve the analysis results is presented in

Section 5.3.4. Finally, we combinate abstract interpretation and data flow analysis techniques in Section 5.3.5 which enables the checking for a large set of generic properties.

5.3.1 Abstract Simulation

Program simulation is a valuable method to track program behavior for debugging or testing. It is the execution of a program for a specific instance of input and program variables. However, when considering input variables the simulation has often to be performed for each single possible input in order to track the different possible behaviors. This is time consuming or even impossible when there are an infinite number of possible inputs. Therefore, it is desirable to test all possible instances at once. We call such a simulation *abstract simulation* which is in fact a form of abstract interpretation. Abstract simulation comprises a number of possible simulation runs but as simulation itself it is not guaranteed to terminate. In the following we refer to abstract interpretation if we include additional acceleration techniques which ensure termination and to abstract simulation if not. Hence, these two terms are close to each other. Nonetheless, for debugging and testing on an abstract level abstract simulation is as valuable as program simulation is on the concrete level.

In this section we define an interval based abstract operational semantics for IL programs which corresponds to the notion of the abstract collecting semantics introduced in Section 3.5.3. Abstract simulation is the program execution based on this abstract semantics. Note, as mentioned in the introduction to the abstract interpretation framework any abstraction includes the possibility of over-approximation. There is no difference for abstract simulation, e.g., when abstracting possible input values $\{1, 2, 4\}$ by the interval $[1, 4]$ the abstract simulation takes also the input 3 into account, which is not part of the original set. For debugging and the testing of safety properties this is often acceptable and still better than testing any possible instance.

As its concrete counter-part in Section 4.4.3 the interpretation of the abstract semantics is based on labeled transition systems where nodes are configurations and the transitions themselves represent the i/o behavior as well as the abstract execution of single IL statements. Each execution of an IL program is then covered by a run in this transition system. We define the notion of *abstract states* and *abstract configurations* as follows:

Definition 5.3 (abstract state)
The global abstract IL state *contains the values of all variables and is modeled as a mapping* $\Sigma^{\#} : Var \cup \{cr\} \rightarrow D^{\#}$*, where* $D^{\#}$ *stands for the abstract data domain.*

Again, we assume the values in the state to be type consistent and use $\sigma^{\#}$ as typical element of $\Sigma^{\#}$.

Definition 5.4 (abstract configuration)
An IL configuration $\gamma : Locs \times \Sigma^{\#} \times Mode$ *of a program is characterized by*

- *a location* $l \in Locs$,
- *an abstract state* $\sigma^{\#} \in \Sigma^{\#}$, *and*
- *a configuration of type Mode, which can be either* I, O *or* $\mathsf{C}(ILi)$, *where ILi is an IL instruction.*

The differences between abstract states or abstract configurations to their concrete counterparts are mainly in the data domains. While on the concrete level we assumed integers and Booleans as the underlying domains we consider the lattice of intervals $\langle \mathcal{I}, \subseteq_{\mathcal{I}} \rangle$ as presented in Example 3.5 and the lattice of Booleans $\langle \mathcal{B}, \subseteq_{\mathcal{B}} \rangle$ as depicted in Figure 5.8.

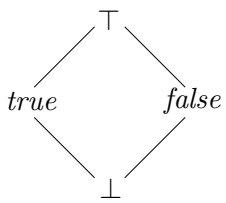

Figure 5.8: Lattice of Booleans

Definition 5.5 (abstract labeled transition system)
With every IL graph G_P *we associate an* abstract labeled transition system $\mathcal{T}_P^{\#} = (\Gamma^{\#}, \gamma_0^{\#}, \rightarrow_{\xi}^{\#})$, *where*

- $\Gamma^{\#}$ *denotes the set of abstract configurations,*
- $\gamma_0^{\#} \in \Gamma^{\#}$ *is the initial configuration and*
- $\rightarrow_{\xi}^{\#}$ *is the transition relation between abstract configurations.*

The initial configuration $\gamma_0^{\#}$ *is given by* $(l_0, \sigma_0^{\#}, \mathsf{I})$, *where the initial state* $\sigma_0^{\#}$ *evaluates all Booleans to* $\top$ *and all integer intervals to top element of the lattice* $[-\infty, +\infty]$. *The operational rules are shown in Figure 5.4–5.9 specifying the labeled transition relation* $\rightarrow_{\xi}^{\#}$ *between system configurations.*

The initial configurations are abstractions of the initial configuration for the concrete level The operational rules a very similar to the ones of Section 4.4.3. However, this time we do not operate on Booleans and integers but on the lattices of Booleans $\langle \mathcal{B}, \subseteq_{\mathcal{B}} \rangle$ and intervals $\langle \mathcal{I}, \subseteq_{\mathcal{I}} \rangle$. Therefore, the standard logical and arithmetic operators do not apply here. Tables 5.3.1 to 5.3.1 define the respective abstract operators. Note that we consider

all operators to be strict, i.e., if any argument is the bottom element of the respective lattice the result yields the bottom element. For the sake of simplicity this is not explicitly mentioned in the definitions.

Since the operational rules of Figure 5.4–5.9 are very close to the ones of Section 4.4.3 we do not explicitly define all operators. In particular the operators augmented by the N qualifier are omitted. An extension to the full set is straightforward. Note, however, that in an abstract semantics comparisons and logic operations might result into an unknown, i.e., $\top$, result. This implies that conditional jumps cannot uniquely be determined. In the rule set this is reflected by non-determinism in the case that the current result is equal to $\top$.

Table 5.1: Abstract Boolean connectivities

operator	abstract semantics
$\neg^{\#}$	$\neg^{\#} b = \begin{cases} \top & \text{if } b = \top \\ \neg b & \text{otherwise} \end{cases}$
$\wedge^{\#}$	$b_1 \wedge^{\#} b_2 = \begin{cases} b_1 \wedge b_2 & \text{if } b_1 \neq \top \text{ and } b_2 \neq \top \\ \top & \text{otherwise} \end{cases}$
$\vee^{\#}$	$b_1 \vee^{\#} b_2 = \begin{cases} true & \text{if } b_1 = true \text{ or } b_2 = true \\ false & \text{if } b_1 = false \text{ and } b_2 = false \\ \top & \text{otherwise} \end{cases}$

Table 5.2: Abstract arithmetic operators

operator	abstract semantics
$+^{\#}$	$i_1 +^{\#} i_2 = [\mathsf{glb}(i_1 + i_2), \mathsf{lub}(i_1 + i_2)]$
$-^{\#}$	$i_1 -^{\#} i_2 = [\mathsf{glb}(i_1 - i_2), \mathsf{lub}(i_1 - i_2)]$
$*^{\#}$	$i_1 *^{\#} i_2 = [\min(product), \max(product)]$ where $product = \{\mathsf{glb}(i_1 * i_2), \mathsf{lub}(i_1 * i_2)\}$

Table 5.3: Abstract comparisons

operator	abstract semantics
$=^{\#}$	$i_1 =^{\#} i_2 = \begin{cases} true & \text{if } i_1 =_{\mathcal{I}} i_2 \\ false & \text{if } i_2 \neq_{\mathcal{I}} i_1 \end{cases}$
$\neq^{\#}$	$i_1 \neq^{\#} i_2 = \begin{cases} true & \text{if } i_2 \neq_{\mathcal{I}} i_1 \\ false & \text{if } i_1 =_{\mathcal{I}} i_2 \end{cases}$
$<^{\#}$	$i_1 <^{\#} i_2 = \begin{cases} true & \text{if } i_1 \subset_{\mathcal{I}} i_2 \\ false & \text{if } i_2 \subseteq_{\mathcal{I}} i_1 \\ \top & \text{otherwise} \end{cases}$
$\leq^{\#}$	$i_1 \leq^{\#} i_2 = \begin{cases} true & \text{if } i_1 \subseteq_{\mathcal{I}} i_2 \\ false & \text{if } i_2 \subset_{\mathcal{I}} i_1 \\ \top & \text{otherwise} \end{cases}$
$>^{\#}$	$i_1 >^{\#} i_2 = \begin{cases} true & \text{if } i_2 \subseteq_{\mathcal{I}} i_1 \\ false & \text{if } i_1 \subset_{\mathcal{I}} i_2 \\ \top & \text{otherwise} \end{cases}$
$\geq^{\#}$	$i_1 \geq^{\#} i_2 = \begin{cases} true & \text{if } i_2 \subset_{\mathcal{I}} i_1 \\ false & \text{if } i_1 \subseteq_{\mathcal{I}} i_2 \\ \top & \text{otherwise} \end{cases}$

Table 5.4: Abstract operational semantics: Mode switches

$$\frac{\sigma^{\#\prime} = \sigma^{\#}_{[\,\vec{x} \mapsto^{\#} \vec{v}\,]} \qquad \vec{x}^{\#} = Var_{in}}{(l, \sigma^{\#}, \mathsf{I}) \rightarrow^{\#}_{?v} (l, \sigma'^{\#}, \mathsf{C}(instr(l)))} \textsc{Input}$$

$$\frac{instr(l) = RET}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l, \sigma^{\#}, \mathsf{O})} \text{RET}$$

$$\frac{\vec{v}^{\#} = [\![\vec{x}]\!]^{\#}(\sigma^{\#}) \qquad \vec{x}^{\#} = Var_{out}}{(l, \sigma^{\#}, \mathsf{O}) \rightarrow^{\#}_{!\vec{v}} (l, \sigma^{\#}, \mathsf{I})} \textsc{Output}$$

Table 5.5: Abstract operational semantics: Basics

$$\frac{instr(l) = LD \qquad \sigma^{\#\prime} = \sigma^{\#}_{[\,op(l)^{\#} \mapsto^{\#} cr^{\#}\,]} \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))}\ \text{LD}$$

$$\frac{instr(l) = ST \qquad \sigma^{\#\prime} = \sigma^{\#}_{[\,cr^{\#} \mapsto^{\#} op^{\#}(l)\,]} \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))}\ \text{ST}$$

$$\frac{instr(l) = S \qquad \sigma^{\#\prime} = \sigma^{\#}_{[\,true \mapsto^{\#} op^{\#}(l)\,]} \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))}\ \text{S}$$

$$\frac{instr(l) = R \qquad \sigma^{\#\prime} = \sigma^{\#}_{[\,false \mapsto^{\#} op^{\#}(l)\,]} \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))}\ \text{R}$$

Table 5.6: Abstract operational semantics: Arithmetics

$$\frac{instr(l) = ADD \qquad \sigma^{\#\prime} = \sigma^{\#}_{[\,cr^{\#} \mapsto^{\#} cr^{\#} +^{\#} op^{\#}(l)\,]} \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))}\ \text{ADD}$$

$$\frac{instr(l) = SUB \qquad \sigma^{\#\prime} = \sigma^{\#}_{[\,cr^{\#} \mapsto^{\#} cr^{\#} -^{\#} op^{\#}(l)\,]} \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))}\ \text{SUB}$$

$$\frac{instr(l) = MUL \qquad \sigma^{\#\prime} = \sigma^{\#}_{[\,cr^{\#} \mapsto^{\#} cr^{\#} *^{\#} op^{\#}(l)\,]} \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))}\ \text{MUL}$$

Table 5.7: Abstract operational semantics: Boolean logics

$$\frac{instr(l) = NOT \qquad \sigma^{\#\prime} = \sigma^{\#}_{[\,cr^{\#} \mapsto^{\#} \neg^{\#} cr^{\#}\,]} \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))}\ \text{NOT}$$

$$\frac{instr(l) = AND \qquad \sigma^{\#\prime} = \sigma^{\#}_{[\,cr^{\#} \mapsto^{\#} cr^{\#} \wedge^{\#} op^{\#}(l)\,]} \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))}\ \text{AND}$$

$$\frac{instr(l) = OR \qquad \sigma^{\#\prime} = \sigma^{\#}_{[\,cr^{\#} \mapsto^{\#} cr^{\#} \vee^{\#} op^{\#}(l)\,]} \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))}\ \text{OR}$$

Table 5.8: Abstract operational semantics: Comparisons

$$\frac{instr(l) = GT \qquad \sigma^{\#\prime} = \sigma^{\#}_{[\, cr^{\#} \mapsto^{\#} cr^{\#} >^{\#} op^{\#}(l)\,]} \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))}\ \text{GT}$$

$$\frac{instr(l) = GE \qquad \sigma^{\#\prime} = \sigma^{\#}_{[\, cr^{\#} \mapsto^{\#} cr^{\#} \geq^{\#} op^{\#}(l)\,]} \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))}\ \text{GE}$$

$$\frac{instr(l) = LT \qquad \sigma^{\#\prime} = \sigma^{\#}_{[\, cr^{\#} \mapsto^{\#} cr^{\#} <^{\#} op^{\#}(l)\,]} \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))}\ \text{LT}$$

$$\frac{instr(l) = LE \qquad \sigma^{\#\prime} = \sigma^{\#}_{[\, cr^{\#} \mapsto^{\#} cr^{\#} \leq^{\#} op^{\#}(l)\,]} \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))}\ \text{LE}$$

$$\frac{instr(l) = EQ \qquad \sigma^{\#\prime} = \sigma^{\#}_{[\, cr^{\#} \mapsto^{\#} cr^{\#} =^{\#} op^{\#}(l)\,]} \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))}\ \text{EQ}$$

$$\frac{instr(l) = NEQ \qquad \sigma^{\#\prime} = \sigma^{\#}_{[\, cr^{\#} \mapsto^{\#} cr^{\#} \not=^{\#} op^{\#}(l)\,]} \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#\prime}, \mathsf{C}(instr(l')))}\ \text{NEQ}$$

Table 5.9: Abstract operational semantics: Jumps

$$\frac{instr(l) = LABEL \qquad l' \in Succ(l)}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#}, \mathsf{C}(instr(l')))}\ \text{LABEL}$$

$$\frac{instr(l) = JMP \qquad l' \in Succ(l) \qquad instr(l') = LABEL}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#}, \mathsf{C}(instr(l')))}\ \text{JMP}$$

$$\frac{\begin{array}{c} instr(l) = JMPC \qquad l' \in Succ(l) \\ cr^{\#}(\sigma^{\#}) = false \vee cr^{\#}(\sigma^{\#}) = \top \qquad instr(l') \neq LABEL \end{array}}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#}, \mathsf{C}(instr(l')))}\ \text{JMPC}_{\text{FF}}$$

$$\frac{\begin{array}{c} instr(l) = JMPC \qquad l' \in Succ(l) \\ cr^{\#}(\sigma^{\#}) = true \vee cr^{\#}(\sigma^{\#}) = \top \qquad instr(l') = LABEL \end{array}}{(l, \sigma^{\#}, \mathsf{C}(instr(l))) \rightarrow^{\#} (l', \sigma^{\#}, \mathsf{C}(instr(l')))}\ \text{JMPC}_{\text{TT}}$$

5.3.2 Abstract Interpretation Applied to Program Analysis

In this section we apply the idea of abstract interpretation to static program analysis. In fact this corresponds closely to abstract simulation as defined in the previous section but includes acceleration techniques for termination.

Abstract interpretation for program analysis is about the computation of program properties, e.g., in our case the computation of all possible program values at a given program node. This means, we like to compute an approximation of the collecting semantics as introduced in Section 3.5.3. This can be stated in terms of computing a (least) fixed point solution of a system of semantic equations

$$f^{\#}: \begin{array}{rcl} \vec{x}_1 & = & \phi_1^{\#}(\vec{x}_1, \ldots, \vec{x}_n) \\ \vdots & & \\ \vec{x}_n & = & \phi_n^{\#}(\vec{x}_1, \ldots, \vec{x}_n) \end{array}$$

where each index i, $1 \leq i \leq n$, represents a program node of an IL graph and each function $\phi_i^{\#}$ is a continuous function from $L^{\#^n}$ to $L^{\#}$ where $L^{\#}$ is the abstract lattice of program properties/information. Each function $\phi_i^{\#}$ computes the abstract property holding at program node i after the execution of one program step from every node leading to i. In particular, considering a set of variables $Var \cup cr$ of types Boolean and integer and the standard abstractions used in this work so far, then $L^{\#}$ is the product space of $\langle \mathcal{B}, \subseteq_{\mathcal{B}} \rangle$ and interval lattices $\langle \mathcal{I}, \subseteq_{\mathcal{I}} \rangle$ representing the abstract values for each program variable.

```
VAR
        x:INT;
END_VAR

        LD 1
        ST x
label:
        LD x
        ADD 1
        ST x
        LE 10
        JMPC label
        RET
```

Figure 5.9: Simple IL example

What does this mean in an IL setting? Consider the short IL program P in Figure 5.9 which successively increases a variable x by 1, starting from

1, until 10 is reached. We consider the IL graph $G_P = (N, E, n_{ini}, n_{fin}, stm)$ representing the control structure of the IL program P. We number the nodes as follows:

```
          [LD 1]^1
          [ST x]^2
[label:]^3
          [LD x]^4
          [ADD 1]^5
          [ST x]^6
          [LE 10]^7
          [JMPC label]^8
          [RET]^9
```

Additionally we have the initial node n_{ini} with its edge to node number 1 and the finial node n_{fin} with its edge from node number 9. We assign the number 0 to n_{ini} and 10 to n_{fin}.

If for each equation of the equation system we take into account only the parameters the equation directly depends on, i.e., the predecessors of a node we obtain the following equation system:

$$\begin{aligned}
\vec{x}_0 &= \phi_0^{\#}(\vec{x}_0) \\
\vec{x}_1 &= \phi_1^{\#}(\vec{x}_0) \\
\vec{x}_2 &= \phi_2^{\#}(\vec{x}_1) \\
\vec{x}_3 &= \phi_3^{\#}(\vec{x}_2, \vec{x}_8) \\
\vec{x}_4 &= \phi_4^{\#}(\vec{x}_3) \\
\vec{x}_5 &= \phi_5^{\#}(\vec{x}_4) \\
\vec{x}_6 &= \phi_6^{\#}(\vec{x}_5) \\
\vec{x}_7 &= \phi_7^{\#}(\vec{x}_6) \\
\vec{x}_8 &= \phi_8^{\#}(\vec{x}_7) \\
\vec{x}_9 &= \phi_9^{\#}(\vec{x}_8) \\
\vec{x}_{10} &= \phi_{10}^{\#}(\vec{x}_9)
\end{aligned}$$

The program information at each node given by $\vec{x}$ are the array of abstract values $\langle cr^{\#}, x^{\#} \rangle$ of the current result and the variable x.

The equations, e.g., $\vec{x}_3 = \phi_3^{\#}(\vec{x}_2, \vec{x}_8)$, should read as follows: "The information about the variables in $\vec{x}$ at node 3 depend on the semantic effect defined by $\phi_3^{\#}$ depending on the information about $\vec{x}$ at nodes 2 and 8". In case of $\phi_3^{\#}$ the semantic effect is the union of the information coming from

node 2 and node 8. For all other program nodes their semantic effect is described by the resulting program values defined by the abstract operational semantics and its operator in Section 5.3.1.

Moreover, $\phi_0^\#$ corresponds to the initialization of the variables in $\vec{x}$ on the abstract level. This means, if there is no value specified in the declarative part, it is either $\top$ or $[+\infty, -\infty]$ depending on its type. This represents the fact that the value is *unknown.* We assume that the current result, which is dynamically typed, is initialized by $\top$. For any type inconsistent operation or union of values of different types the outcome is the bottom element $\bot$.

A *solution* to the equation system can be:

$$\begin{array}{lllllll}
\vec{x}_0 & = & \phi_0^\#(\vec{x}_0) & = & \langle \top & , & [-\infty, +\infty] \rangle \\
\vec{x}_1 & = & \phi_1^\#(\vec{x}_0) & = & \langle [1,1] & , & [-\infty, +\infty] \rangle \\
\vec{x}_2 & = & \phi_2^\#(\vec{x}_1) & = & \langle [1,1] & , & [1,1] \rangle \\
\vec{x}_3 & = & \phi_3^\#(\vec{x}_2, \vec{x}_8) & = & \langle \bot & , & [1,9] \rangle \\
\vec{x}_4 & = & \phi_4^\#(\vec{x}_3) & = & \langle [1,9] & , & [1,9] \rangle \\
\vec{x}_5 & = & \phi_5^\#(\vec{x}_4) & = & \langle [2,10] & , & [1,9] \rangle \\
\vec{x}_6 & = & \phi_6^\#(\vec{x}_5) & = & \langle [2,10] & , & [2,10] \rangle \\
\vec{x}_7 & = & \phi_7^\#(\vec{x}_6) & = & \langle \top & , & [2,10] \rangle \\
\vec{x}_8 & = & \phi_8^\#(\vec{x}_7) & = & \langle \top & , & [2,10] \rangle \\
\vec{x}_9 & = & \phi_9^\#(\vec{x}_8) & = & \langle false & , & [10,10] \rangle \\
\vec{x}_{10} & = & \phi_{10}^\#(\vec{x}_9) & = & \langle false & , & [10,10] \rangle
\end{array}$$

In fact, this solution is the *best solution*, i.e., is the least fixed point of the equation system.

As argued in Section 3.5 it is generally not ensured that any iteration strategy solving these equations will terminate. We introduced the concept of widening, which has to be applied here. These means for a chosen subset of equations, we replace each equation by

$$\vec{x}_i = \vec{x}_i \nabla \phi_i^\#(\vec{x}_1, \ldots, \vec{x}_n)$$

where ∇ is the widening operator. We apply the widening operator pointwise to each component of $\vec{x}$ and interpret it for integers as defined in Example 3.6:

$$\begin{array}{rcl}
[] \nabla x = x \nabla [] & = & x \\
[a_1, b_1] \nabla [a_2, b_2] & = & [\text{if } a_2 < a_1 \text{ then } -\infty \text{ else } a_1, \\
 & & \text{if } b_2 > b_1 \text{ then } +\infty \text{ else } b_1].
\end{array}$$

and for Booleans we define it as:

$$\begin{array}{rcl}
\bot \nabla b = b \nabla \bot & = & b \\
b_1 \nabla b_2 & = & \text{if } b_1 \neq b_2 \text{ then } \top \text{ else } b_1.
\end{array}$$

Note, a widening operator for Booleans is not really necessary since the lattice of Booleans is of finite hight. Moreover, the above definition of Boolean widening corresponds to the definition of Boolean union, however, we introduce this widening operator also for Booleans, in order to keep the framework as consistent and as simple as possible.

In order to compute the least approximated fixed point (by widening) we have to iterate

$$\begin{aligned} \mu^{\nabla}_{f^{\#}_0} &= \bot \\ \mu^{\nabla}_{f^{\#}_{n+1}} &= f^{\#}(\mu^{\nabla}_{f^{\#}_n}) \end{aligned}$$

as defined in Section 3.5. However, there remain two questions open:

1. What is a good iteration strategy?
2. What is a good set of widening points?

The answer to question number 1 is crucial to obtain an efficient method for fixed point iteration. Especially, if the equation are loosely coupled as in the example of Figure 5.9 a lot of time can be saved, if the number of equation evaluations which "do not produce something new" are minimized.

A good set of widening point is as small as possible, since any widening most likely will lead to a loss of precision. Moreover, the demand for a good iteration strategy is closely related to widening points. Any iteration strategy should apply widening as less as possible and as late as possible to reduce the chances of unnecessary over-approximations. These two issues are discussed in more detail in the next section.

5.3.3 Efficient Fixed Point Computation

Data flow analysis as well as abstract interpretation rely significantly on the computation of fixed points. Simply because the fixed points in these settings represent a stabilization of information propagation which is the desired state. The iterations of fixed points are also the most time consuming part in static program analysis. Hence, an efficient iteration strategy is desireable.

Any fixed point iteration in static program analysis can be viewed as a fixed point iteration of a finite equation system, which describes the data dependencies between the different nodes of a data flow graph. We say an interation strategy A is *more efficient* than another strategy B if the number of equations which have to be evaluated with A are less than with B. In the following we describe several iteration startegies. A more comprehensive overview can be found in [Sch95].

Chaotic Strategies

We already introduced in Section 3.4.4 the notion of chatoic iteration. In fact, chaotic iteration is just a random evaluation of equations. Often a notion of *fairness* is required, this means each equation will always enventually be evaluated, i.e., is never neglected infinitely.

Numerical analysis suggest two more advanced variants of chaotic itertations: *Jacobi iteration* and *Gauss-Seidel itertation* strategy. Devloped to find solutions in sets of linear equations they provide effective means for fixed point iteration in our context. Although the Gauss-Seidel itertation strategy is in general slightly more efficient since in each iteration the computed values are immediately re-used, if possible, they to not cover the *structure* of the problem, e.g., the information flow in static program analyis, and, therefore, cannot be considered as the best choice.

Worklists

A better strategy is to order the equations to be evaluated according to the structure of the problem. This gave rise to methods using *worklists.* A worklist is a list of equations (intitially all) where subsequently one equation is taken (and removed), evaluated and depending on the underlying decision algorithm on the worklist the equation is added again or not. A fixed point is reached if the worklist ist empty.

There have been different approaches on how to implement a worklist and its underlying decision algorithm. Two variants are *workstacks* where equations are treated in a LIFO manner and *priority queues* where equations are evaluated in reverse postorder.

Although worklists seem to work fine for acyclic graphs, it is not easy to find a good underlying decision procedure anymore if it comes to cylic graphs.

Weak Topological Order

Global static program analysis often operates on cyclic graphs, i.e., equation systems with cyclic dependencies, which originate from loops in the programs. Iteration strategies should refelct these cycles and treat them in an appropiate way.

One possibility to group cycles are is given by *strongly connected components* (SCC). In a strongly connected component every node of graph in this component can be reached from any other node in the same component by following the vertices, i.e., by a path given by the edge relation. Tarjan developed an efficient algorithm to obtain all SCCs from a given graph (cf. [LT79]).

As itertation strategies there are a number of suggestion in which order to visit these SCCs as well as domain independent dynamic optimisation for

orders within each SCC. These startegies comprise methods where worklist are used inside the SCC sometimes combining reverse postorder traversals with workstacks. The workstacks are optimised by certain criteria such as youngest vertex first, depth first, or direct or indirect dependencies just to mention some.

An ingenious idea was presented by Bourdoncle by the introduction of *weak topological orders* (WTO) [Bou93] in the context of abstract interpretation. The key idea is to hierarchically decompose a directed graph into strongly connected components and subcomponents by the recursive application of Tarjan's algorithm. Subcomponents of an SCC are identified by removing the *head* w of an SCC as well as all vertices pointing to w and applying Tarjan's algorithm once more. The head is determined though the algorithm and is, generally spoken, the connecting node/program statement of a loop. The advantage of Bourdoncle's algorithm is that it can be computed statically before the iteration process while at the same time it reduces the number of widening application for abstract interpretation. The heads exactly correspond to the widening points.

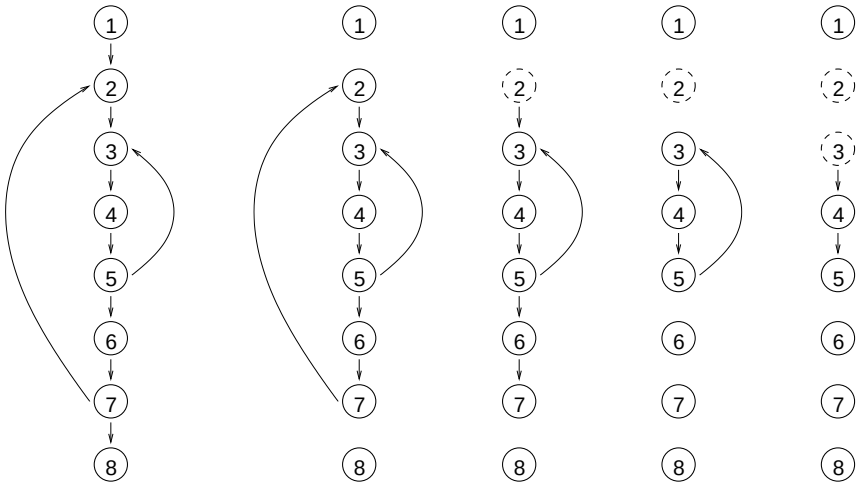

Figure 5.10: Application of WTO algorithm

We illustrate the algorithm for constructing a WTO by a graph with eight nodes and two loops as depicted in Figure 5.10. This first graph from the left is the original which represents the information flow in a program which might correspond to the control flow. The second graph is a decomposition into SCCs. We have the trivial SCCs 1 and 8 as well as the non-trivial SCC of nodes 2 to 7. In the third graph node number 2 is identified as the head (dashed circle) as determined by the WTO algorithm and the back edge to the head is removed. The result of more decompositions into srongly connected sub-components is shown in the fifth graph, the nodes numbered 3 to 5 constitute a non-trivial sub-SCC. Its head is node number 3 as shown in

the sixth graph. Any more application of Tarjan's algorithm does only result into trivial SCCs. Hence, we have two non-trivial SCCs: nodes 2 to 7 and 3 to 5 with their respective heads 2 and 3. Moreover, the second SCC is nested in the first one which can be denoted using parentheses: 1 (**2** (**3** 4 5) 6 7) 8.

As pointed out in [Sch95] WTOs appear to be the structure of choice for fixed point computation in static program analysis, although [Edw97] showed they are not necessarily optimal.

Iteration Strategies Weak topological orders provide a structure for fixed point operation, however, there are at least two options for iteration stragtegies: a *recursive* and an *itertative* strategy. For both strategies stabilisation is only tested at each component head, i.e., whenever all component heads have stabelised a fixed point is reached.

For the recursive itertaion strategy the equations determined by the edge relation are evaluated recursively to the nested sequence of components and sub-components. This means whenever a sub-component C' is found in an SCC C, the sub-component C' is stabilized first, before stabilizing the original component C. In contrast, in the iterative strategy every outermost component of a WTO is stabelized, thereby removing the sub-component structure without loosing the order structure.

In the example of Figure 5.10 this means that the recursive startegy first stabelized the SCC 3 to 5 before stabelizing 2 to 7. Note, however, that that stabilizing 3 to 5 might be repreated several times depending on the outer component. The iterative strategy stabilizes the SCC 2 to 7 where the order provided by the WTO is taken into accound until all heads are stable.

As pointed out in [Bou93] the recursive startegy is in general computaionally more efficient, however, this depends much on the actual graph structure to be analysed. Nonetheless, in this work we us WTOs with a recursive iteration strategy.

5.3.4 Precision Impovement for Non-Relational Abstractions

As described in Section 5.3.3 good iteration strategies and sets can be obtained through WTOs. Although the widening points are carefully choosen, the over-approximation is in many cases immense. In this section we reason that standard narrowing techniques to reduce this over-approximation are hard to apply and better methods for enhancing the precision of the abstract interpretation procedure are desired. Based on this consideration we present a heuristics based on a constraint solving approach to achieve this goal.

The Dilemma

Consider again the program of Figure 5.9. The WTO structure for this program based on its IL graph is:

$$0\ 1\ (\mathbf{2}\ 3\ 4\ 5\ 6\ 7\ 8)\ 9\ 10$$

which means node number 2 is the single widening point. If we apply the recursive iteration strategy with the standard widening techniques to the program, we obtain the following abstract interpretation result:

$$\begin{array}{lcllcl}
\vec{x}_0 & = & \phi_0^{\#}(\vec{x}_0) & = & \langle \top & , \ [-\infty, +\infty] \rangle \\
\vec{x}_1 & = & \phi_1^{\#}(\vec{x}_0) & = & \langle [1,1] & , \ [-\infty, +\infty] \rangle \\
\vec{x}_2 & = & \phi_2^{\#}(\vec{x}_1) & = & \langle [1,1] & , \ [1,1] \rangle \\
\vec{x}_3 & = & \vec{x}_3 \nabla \phi_3^{\#}(\vec{x}_2, \vec{x}_8) & = & \langle \bot & , \ [1,+\infty] \rangle \\
\vec{x}_4 & = & \phi_4^{\#}(\vec{x}_3) & = & \langle [1,+\infty] & , \ [1,+\infty] \rangle \\
\vec{x}_5 & = & \phi_5^{\#}(\vec{x}_4) & = & \langle [2,+\infty] & , \ [1,+\infty] \rangle \\
\vec{x}_6 & = & \phi_6^{\#}(\vec{x}_5) & = & \langle [2,+\infty] & , \ [2,+\infty] \rangle \\
\vec{x}_7 & = & \phi_7^{\#}(\vec{x}_6) & = & \langle \top & , \ [2,+\infty] \rangle \\
\vec{x}_8 & = & \phi_8^{\#}(\vec{x}_7) & = & \langle \top & , \ [2,+\infty] \rangle \\
\vec{x}_9 & = & \phi_9^{\#}(\vec{x}_8) & = & \langle false & , \ [2,+\infty] \rangle \\
\vec{x}_{10} & = & \phi_{10}^{\#}(\vec{x}_9) & = & \langle false & , \ [2,+\infty] \rangle
\end{array}$$

This result grossly over-approximates the best solution. There is no longer any upper bound on x in most cases. In particular, the information that the jump operation depends on whether x is less than 10 or not is lost. The reason is the following: After a certain number of iterations and the application of the widening operator we arrive at a state where

$$\vec{x}_8 \ = \ \phi_8^{\#}(\vec{x}_7) \ = \ \langle \top \ , \ [1,+\infty] \rangle.$$

However, if we use this information to solve the dependent equations

$$\begin{array}{lcl}
\vec{x}_3 & = & \vec{x}_3 \nabla \phi_3^{\#}(\vec{x}_2, \vec{x}_8) \text{ and} \\
\vec{x}_9 & = & \phi_9^{\#}(\vec{x}_8)
\end{array}$$

it is clear that there is no way to directly infer that the value of x is either below or above 10. One reason is that we use a non-relational abstraction, namely intervals, and relate them in such a way that we cannot connect information gathered partially over several steps.

Although in a forward iteration using widening such a first over-approximation is not uncommon, often in a second phase with a backward iteration and standard narrowing this problem can be remedied. This is, unfortunately, not the case for IL programs where each single line of code is examined in the way presented above. The problem remains the same, we are unable to propagate partial information (of jump conditions).

A Constraint Solving Heuristic

Every computation of an IL program takes place at the current result cr. Therefore, to make use of the information gathered during the computation process concerning the current result, we record its relevant history. This information is then used to constrain the possible values of the program variables, e.g., at jump labels.

History expressions. We include the history h_{cr} of the current result by representing it as an expression involving constants and variables. This can be done completely statically in a one pass forward analysis. After each statement, the expression of the history h_{cr} is updated according to the following rules:

- If the statement directly follows a conditional jump, the expression is the negation of the expression at the jump. If the statement directly follows a negated conditional jump, the expression is the same as the expression at the jump.
- If the statement is a label, the expression is *unknown.*
- If the statement is a load (`LD`) or store (`ST`) instruction the expression is the operand.
- In any other case the expression is updated according to the statement.

The rule set ensures that we always record the most recent history, i.e., the history from the last load or store statement. However, it is possible to extend histories further backward to capture more complex effects. We assume the history at jump labels to be unknown, because there might be several jumps to the same label which makes a static determination of the possible expression for the label more complicated. However, in the analysis and constraining process of the information accumulated at the labels, all possible jumps to that label are taken into account.

If we apply the set of rules to the program of Figure 5.9 we obtain the result presented in Figure 5.11.

The history expressions record the most recent information about program and i/o variables involved in the computation. They record the current state of computation.

Constraint based analysis. The history expressions help to gather more information at each program node than there was available in the original setting. In particular, at any jump there is now explicitly the jump condition available. This condition splits the possible program values depending on whether the jump is taken or not. For instance in the example of Figure 5.11

```
VAR
        x:INT;
END_VAR

        LD 1        h_cr=1
        ST x        h_cr=x
label:              h_cr=unknown
        LD x        h_cr=x
        ADD 1       h_cr=x+1
        ST x        h_cr=x
        LE 10       h_cr=x<10
        JMPC label  h_cr=x<10
        RET         h_cr=¬(x<10)
```

Figure 5.11: IL program augmented by history expressions

the condition $\mathtt{x} < 10$ splits the state abstract space of x into all values less than 10 and all values greater or equal to 10.

The result of constraining a vector of variables $\vec{x}$ is denoted by $\vec{x}^c$. Every variable not effected by the constraints remain unaltered. The resulting constrained vector at each label is the *union* of every constrained vector from any jump to this label. Taking the union ensures a worst case approximation, since in a worst case the information at a label be propagated from any jump to that specific label. A more refined way is to determine the reachable code beforehand and then take into account only constraints for reachable jumps to the specific label.

We alter the original abstract interpretation procedure by constraining each label by the set of constrained values leading to that label. This means we have

$$\vec{x}_i = \vec{x}_i \cap \phi_i^{\#c}(\vec{x}_1^c, \ldots, \vec{x}_n^c)$$

where $\phi_i^{\#c}$ is just the union of the constrained values. Respectively, for widening points we extend the framework to

$$\vec{x}_i = (\vec{x}_i \nabla \phi_i^{\#}(\vec{x}_1, \ldots, \vec{x}_n)) \cap \phi_i^{\#c}(\vec{x}_1^c, \ldots, \vec{x}_n^c).$$

Similarly, each node after a (negated) conditional jumps is constrained as well. This time by the constrain associated to its node.

$$\vec{x}_i = \vec{x}_i \cap \vec{x}_i^c.$$

If we apply this to the example of Figure 5.11 we obtain the two altered equations:

$$\begin{aligned} \vec{x}_3 &= (\vec{x}_3 \nabla \phi_3^{\#}(\vec{x}_2, \vec{x}_8)) \cap \phi_3^{\#c}(\vec{x}_2^c, \vec{x}_8^c) \\ \vec{x}_9 &= \phi_9^{\#}(\vec{x}_8) \cap \vec{x}_9^c. \end{aligned}$$

where $\phi_3^{\#c}(\vec{x}_2^c, \vec{x}_8^c)$ denotes the union of the constrained abstract state spaces as resulted from node 2 and node 8. In the same way $\vec{x}_9^c$ denotes the constrained state space as determined by node 9. In both cases the constrained information is intersected with the result of the widening, i.e., reduces the effect of over-approximation. This results into the following analysis results:

$$
\begin{array}{lcll}
\vec{x}_0 & = & \phi_0^{\#}(\vec{x}_0) & = \langle \bot \quad , \quad [-\infty, +\infty] \rangle \\
\vec{x}_1 & = & \phi_1^{\#}(\vec{x}_0) & = \langle [1,1] \quad , \quad [-\infty, +\infty] \rangle \\
\vec{x}_2 & = & \phi_2^{\#}(\vec{x}_1) & = \langle [1,1] \quad , \quad [1,1] \rangle \\
\vec{x}_3 & = & (\vec{x}_3 \nabla \phi_3^{\#}(\vec{x}_2, \vec{x}_8)) \cap \phi_3^{\#c}(\vec{x}_2^c, \vec{x}_8^c) & = \langle \bot \quad , \quad [1,9] \rangle \\
\vec{x}_4 & = & \phi_4^{\#}(\vec{x}_3) & = \langle [1,9] \quad , \quad [1,9] \rangle \\
\vec{x}_5 & = & \phi_5^{\#}(\vec{x}_4) & = \langle [2,10] \quad , \quad [1,9] \rangle \\
\vec{x}_6 & = & \phi_6^{\#}(\vec{x}_5) & = \langle [2,10] \quad , \quad [2,10] \rangle \\
\vec{x}_7 & = & \phi_7^{\#}(\vec{x}_6) & = \langle \top \quad , \quad [2,10] \rangle \\
\vec{x}_8 & = & \phi_8^{\#}(\vec{x}_7) & = \langle \top \quad , \quad [2,10] \rangle \\
\vec{x}_9 & = & \phi_9^{\#}(\vec{x}_8) \cap \vec{x}_9^c & = \langle false \quad , \quad [10,10] \rangle \\
\vec{x}_{10} & = & \phi_{10}^{\#}(\vec{x}_9) & = \langle false \quad , \quad [10,10] \rangle
\end{array}
$$

The result by the improved method using constraints greatly enhances the precision of the analysis result. In fact, it equals the optimal approximation. However, this is not guaranteed.

Remarks. Although in the example above the optimal approximation is obtained, the use of constraints can only be a heuristic to improve analysis results. In general, solving arbitrary constraints is undecidable. Moreover, if more variables and more complex equations are involved the constraint solving approach suffers precision. There are several ways to reduce this effect. The history expression might be designed to incorporate longer histories, i.e., to reach back over more than one load or store statement and constraints themselves might be solved in an approximated manner. This is, however, not part of this work and is considered for future research.

5.3.5 Static Analysis

In this section we present a number of generic goals for the verification of IL programs together with techniques to check them statically. These checks are mostly based on a combination of the aforementioned abstract interpretation and data flow analysis techniques. The checks below are done by examining the program code line by line if not otherwise stated.

Some checks broaden spectrum we previously discussed, e.g., we take into account variables of types other than Booleans and integers, or we refer to instructions not yet discussed but which are part of the full IL language. New instructions or types should be clear where mentioned. We feel the

broadening gives additional insight into the applicability of the proposed methods.

We classify the outcome of the checks according to their precision and relevance for program correctness. Some results might be imprecise due to (over-)approximation. However, since we are always conservative in the analysis we rather issue a warning than miss an error. Some checks might exhibit defects in the program which are not safety critical, i.e., immediately repeated load or store instructions, in these cases either a warning or remark is issued, depending on their relevance. We give the following classification:

- *errors*, denote that there definitely is an error,
- *warnings*, denote that there is the possibility of an error, but we cannot be sure because of information loss due to abstraction,
- *okays*, denote that we can be sure that a certain statement will not produce any run-time error for the property checked, and
- *remarks*, which indicate that the IL code can be optimized in some way.

In the following the generic verification goals as well as their checking methods and the classification of the outcome are described.

Checks of Assignments and Arithmetic Operations

The following checks make use of the results obtained by the abstract interpretation process.

Overflow/underflow. We check whether an operation violates maximal integer bounds. Violating means that, e.g., a subtraction with a positive value takes place on variables already approximated by $-\infty$ to their lower bound or addition to an upper bound of $+\infty$. In these cases warnings are issued. Moreover, a possible check is to verify that operations do not exceed the defined range for, e.g., the multiplication of two integers is still in the program defined range for integers.

Division by zero. The abstract interpretation results are used to determine whether a division by zero occurs or not. Therefore, program code is examined at the operators `DIV` and `MOD` which might provoke the arithmetic run-time error. We distinguish between constants and variables as operands: if the operand is a constant then an error is issued if the constant is 0 and an okay otherwise. If the operand is a variable a we take a look at the abstract value of a. If it is $[0, 0]$ an error is issued, if 0 is contained in the none-singleton abstract interval of a, a waring is issued. In all other cases an okay is issued.

Array bound checking. Since the full language of IL supports much more complex structures such as arrays, we briefly comment on this. Checks have to concern the indices used for array access. Basically, this is the same as range checking for any other variable. The variable used for the index has to remain within its predefined range. In particular this means that when accessing the array the value of the index variable has to be within the range of the given array size. For this check the abstract interpretation framework has to be extended to arrays. The method remains the same.

Example 5.2 Consider the following analysis fragment:

```
                    cr# = ⊤, x# = [−∞, +∞]
LD x
                    cr# = [−∞, +∞], x# = [−∞, +∞]
ADD 1
                    cr# = [−∞, +∞], x# = [−∞, +∞]
                    (* WARNING: CR overflow *)
DIV 0
                    cr# = ⊥, x# = [−∞, +∞]
                    (* ERROR: division by zero *)
```

Invariant Conditional Jumps

If the exact Boolean value of $cr^{\#}$ is known at a conditional jump (`JMPC`, `JMPCN`) after the abstract interpretation process, the program will either always or never jump at this program point. We define this as:

Definition 5.6 (invariant conditional jump)
A conditional jump is called invariant *if its jump condition is either always true or always false.*

If a conditional jumps is invariant, it is useless and reveals a possible flaw. A warning is issued.

Example 5.3 Consider the following analysis fragment:

```
                    cr# = [1, 4], x# = [−∞, +∞]
LD 1
                    cr# = [1, 1], x# = [−∞, +∞]
LE 4
                    cr# = true, x# = [−∞, +∞]
JMPC label
                    (* WARNING: jump always 'true' *)
```

It is obvious that the conditional jump `JPMC` always branches to `label`. This might exhibit an undesired behavior, at least the whole program fragment can be substituted by `JMP label`.

Unreachable Code Detection

By unreachable code we refer to programming code which will never be reached by any program execution. In terms of IL language, this means, there are (conditional) jumps that prevent the control flow from sequentially executing every line of code and instead always skip some lines. Hence, these lines of code will never be executed.

There are two possibilities for unreachable code: On the one hand there is simply a combination of `JMP` operators such that some lines are excluded from program execution and on the other hand there are some invariant `JMPC` or `JMPCN` operations producing the same effect.

For a given program P with its IL graph $G_P = (N, E, n_{ini}, n_{fin}, stm)$ the problem can be stated as a simple reachability task in a data flow setting:

> Determine for every program node $n \in N$, if n may be reachable from n_{ini}.

This leads to a forward directed data flow analysis on the lattice $\langle \mathcal{P}(N), \subseteq \rangle$ with the top element $\emptyset$ where the greatest fixed point is computed. The abstract interpretation results are taken into account, this means in contrast to a standard reachability analysis information about invariant conditional jumps are considered. This is a semantic based data flow analysis which might improve the precision of the results significantly. As a result all nodes not in the reachable set are issued in a warning.

Example 5.4 Consider the data flow graph of Figure 5.12 and assume there is a conditional jump that is based on testing whether a variable x is less than 10 or not.

By means of standard data flow analysis every block in the depicted flow graph would be reachable. However, if we take into account a previous abstract interpretation analysis of the respective program which yields that before the conditional jump x is always less than 10, i.e., $x^{\#} = [1, 9]$, then we can deduce that block $B3$ is not reachable.

This shows that abstract interpretation can add precision to standard data flow analysis. In fact, it leads to a semantics based analysis in contrast to the standard semantics independent analysis.

Infinite Loop Detection

Infinite loops pose a severe threat to PLC based control. In general the PLC operating system demands that programs terminate within a certain amount of time. If there is an infinite loop in the program, i.e., it might not terminate, there can be an abortion of the program or even a shutdown of the whole system invoked by the operating system. Thus, it is crucial to detect or rule out infinite loops.

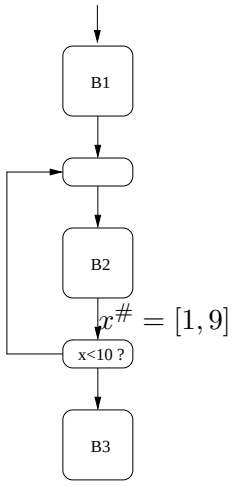

Figure 5.12: Unreachable code and infinite loop

To detect infinite loops it is helpful to analyze the topological structure of loops in the program. This is exactly what has been done by determining the WTO for the abstract interpretation process. WTOs represent a hierarchy of strongly connected component. Hence, in a purely structure based analysis a sub-SCC S is an infinite loop, if the last element of S is a `JMP` instruction and there are no jumps or conditional jumps within this S with a target outside S, i.e., there are no jumps leaving this SCC. If we take into account the results of the abstract interpretation process, the above definition can be relaxed towards: The last element might be an invariant jump (jumping always into S) and there might be invariant conditional jumps within S which always remain within the SCC S.

Example 5.5 Consider again the data flow graph of Figure 5.12. The loop embracing block $B2$ can be determined as an infinite one: Due to the abstract interpretation result, we know that there is an invariant conditional jump, which always jumps back and there is no jump leaving block $B2$.

Type Checking

For the sake of simplicity we previously assumed that all programs are well-typed. However, this cannot be assumed in general. Hence, each access to a variable or the current result *cr* must be checked against well-typedness. Moreover, the current result is dynamically typed, i.e., it can take Booleans as well as integers. Any typing mistake is supposed to be a definite program error.

Fortunately, type checking IL programs comes for free in our abstract interpretation setting. Remember, that any typing mistake, i.e., none-defined

operation like adding a value to a Boolean leads to a $\perp$ value in the abstract interpretation process. Moreover, all arithmetic and Boolean operations are strict, hence, the typing mistake is propagated throughout the program (until a valid `LD` operation is reached). However, we do not want to issue a typing mistake for all subsequent instruction neither do we want to issue a typing mistake for labels, since this is not an operation in itself and it is common that at this merging point from various jump nodes the current result "arrives" with different types. This is not considered a mistake, however, if the current result is used in, e.g., an arithmetic operation in the next instruction it results into an error, which will be issued.

We suggest the following on-the-fly type checking procedure during the abstract interpretation process:

Algorithm:

1. Whenever the first time an operation (not a label) leads to an abstract $\perp$ value we issue a typing error.

2. There is a store operation writing a value v unequal $\perp$ to a constant or to a variable unequal the type of v, then an error is issued.

3. Subsequent errors are not thrown until the abstract value of cr is unequal $\perp$. From that moment on we continue as in 1 and 2.

Item number 1 ensures that the first typing mistake is reported and not all subsequent ones based on that particular one. Item number 2 cares about well-typed store operations (the semantics does not implicitly take care of this) and item number 3 determines when a typing error should no longer be regarded as one dependent on a previous mistake.

In same cases when there are additional flaws in the program (certain types of redundant code etc.) it might be necessary to restart the type checking after correcting the previously detected typing errors in order to find errors previously assumed as follow-ups. A restart of the checks after correction of errors should, however, always be done.

Example 5.6 Consider the following analysis fragment:

```
                    cr# = ⊤, x# = [−∞, +∞]
LD x
                    cr# = [−∞, +∞], x# = [−∞, +∞]
OR true
                    cr# = ⊥, x# = [−∞, +∞]
                    (* ERROR: CR type mismatch *)
MUL 0
                    cr# = ⊥, x# = [−∞, +∞]
LD x
                    cr# = [−∞, +∞], x# = [−∞, +∞]
MUL 0
                    cr# = [0, 0], x# = [−∞, +∞]
ST true
                    cr# = [0, 0], x# = [−∞, +∞]
                    (* ERROR: writing on constant *)
```

Redundant Code

Useless jumps. A jump statement (`JMP`, `JMPC`, `JMPCN`) which jumps right to the next statement is useless. A remark is should be issued.

Dead code. These problems correspond directly to a live variable analysis as defined in Section 3.4. In fact, we are not interested in live but rather the opposite "dead" variables. Again, abstract interpretation helps to reduce the number of possible paths for live variables analysis. We do not go into details about possible variations in dead code analysis here but refer to Section 3.4 and literature about compiler construction such as [Muc97, ASU86]. In any case, a remark should be issued.

Useless statements. There are various combinations of statements which do not make sense, and the a remark is issued:

Each load statement (`LD`, `LDN`) should be preceded by a store statement (`ST`, `STN`, `S`, `R`) or a conditional jump (`JMPC`, `JMPCN`); if it is not, the code before the load statement is unused, since the old value of cr is discarded without having influenced variables or the program flow.

Between two store statements to the same variable there should be some operations modifying cr.

Remarks

In this section we presented a number of possible checks for the verification of IL programs. However, the list is in no sense complete and can be adjusted by the given methods to personal needs. We like to stress, however, that any of these checks can be done completely automatic on the IL source code. Human interaction is only needed to interpret the warnings/errors

and removing them. This makes this approach applicable to IL programmers and does not require further expertise in computer science or thinks alike.

Chapter 6

Tools and Case Studies

In this chapter we show the applicability of the developed verification techniques developed by a number of case studies. They have been provided by academic and industrial partners and cover industrial processes mainly in the field of chemical engineering. In each case the PLC program code for the whole or selected parts of the control process has been provided as SFCs or IL programs.

To apply the introduced verification techniques to the PLC software, we developed two different tools: *SFCheck* and *Homer*. SFCheck performs the translation of SFC source code to the native language of CaSMV and provides additional checking methods, e.g., for safe SFCs. Homer performs the abstract interpretation for IL programs. Both tools are written in the functional language OCaml [CMP00] which is an object-oriented version of ML.

In Section 6.1 we give an introduction to the verification tools developed. The subsequent Sections 6.2 to 6.4 cover various case studies for the considered PLC languages. Note, however, that we presented the first case study in the context of Section 5.2.2. The same tools as described have been used for that one.

6.1 SFCheck/S7Check and Homer

SFCheck is a command line tool to analyze SFCs. It comprises all the verification techniques for SFCs presented in this work and has some additional features as outlined below.

The tool is completely written in the functional object-oriented language OCaml [CMP00]. While SFCheck supports the verification of SFCs given in the standard textual representation, the tool *S7Check* supports the verification of SFCs written in S7, the Siemens notation for SFCs. Since Siemens is one of the world's major supplier of PLCs it is sensible to have a tool to cope with with these SFCs. Both tools have exactly the same functionality.

Thus, for the remainder we refer to SFCheck only, but everything holds for S7Check as well.

The tool has the following features: It automatically translates IEC 61131-3 compliant SFCs to the input language of CaSMV, it supports hierarchical SFCs, action qualifiers, and guards which can also reason about step activities. Moreover, it detects input variables automatically. Various static aspects are analyzed. It checks whether SFCs are safe, steps in transitions are declared, all defined steps belong to some transition, there are unused actions, or there are undefined actions.

Currently, this is a stand alone tool but since it is based on the standard textual representation of SFCs it might well be used as a back-end for various development tools. SFCheck can analyze systems structures or perform model-checking for generic properties, such as reachability of every step, in the background without interaction of the developer. Warnings are issued to the developer if basic requirements appear to be violated.

The tool *Homer* implements the abstract interpretation process for IL programs as well as basic analysis techniques such as syntax checking and proper use of variables. I.e., type-checking and the use of all declared variables and vice versa, if all used variables are declared etc. It is a command line tool operating directly on the IL source code.

The current state of the tool does not include the precision improving techniques as outlined in Section 5.3.4. However, we did use the constraint solving library FaCiLe [fac] for OCaml to enable a subsequent analysis of the case study in Section 6.3.

6.2 A Brick Sorter

The following case study[1] represents a plant sorting certain objects by certain criteria. It has been built as a PLC driven LEGO model and is an abstraction from an industrial plant used for illustration purposes at the University of Nijmegen.

6.2.1 The Plant Layout

The task of the plant is to sort a stack of objects by certain criteria into two groups. Physically, these objects are transported separately from a stack on a conveyor belt to a scanning device installed overhead the conveyor belt. The scanning device determines to which of the two groups the scanned object belongs and passes this information to a sorter at the end of the conveyor belt. This sorter moves the objects either to its left or its right, depending on the information from the scanner. Afterwards they undergo some post-processing steps. A side view of the plant is depicted in Figure 6.1.

[1] Provided by Angelika Mader.

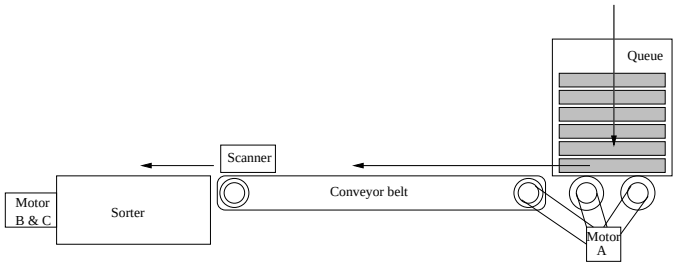

Figure 6.1: Plant from the side

The different devices used in the sorting process are shown in the top view of Figure 6.2. From a queue the objects are transported on the conveyor belt which is driven by a motor A. Moreover, close to the end of the belt there is a scanning device S and a light source L opposite to S in order to create a stable environment for the scanner. The subsequent sorter consists mainly of two rotating forks. The forks are are driven by two motors (B and C). They move the objects to either side of the sorter according to the scanning information.

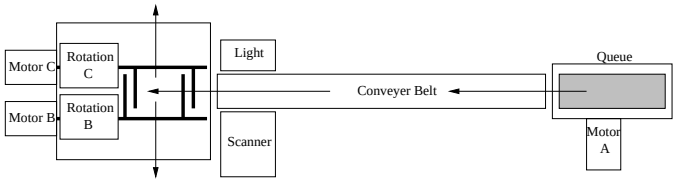

Figure 6.2: Plant from the top

This plant has been build for experimental purposes as a LEGO model at the University of Nijmegen. In this model, the plant sorts LEGO bricks by color, i.e., the stack is sorted into groups of yellow and blue bricks. The whole plant is driven by a PLC and the control programs are given as SFCs and instruction list programs. More precisely, the overall control structure is determined by a SFC where the lower level functionality such as determining the color of a brick from the scanner information is implemented as actions of the SFCs written in instruction list.

For sake of brevity we do not show the whole control part here. The main control SFC consists of seven processes running in parallel. Three of them control the conveyor belt according to information from the sorter and the scanning device. One process controls the scanner and the light source, one initiates a safe shutdown if all bricks are processed, and two processes do the actual sorting. The (simplified) sub-SFC for the sorting process of blue bricks is depicted in Figure 6.3. Starting from the initial step control waits in

`empty63` until all bricks are processed or there is a brick in the scanner, the conveyor belt is running and the sorter is not yet full. Subsequently, another scanning is performed and if the result yields that the brick is blue and the belt is still running the following actions (written in IL) are performed: Some preparing operations of the sorter are done (`sort_blue`), the motor B is switch on and the rotation starts. If at least one full rotation has been performed the sorter is switched off, and the whole process can start all over again. The sorting program for the yellow bricks is similar, but is coupled to motor C. Note that this SFC is comparatively simple but the whole controlling process is complex, since this SFC interacts six other SFCs in parallel.

6.2.2 Verification

For the case study above, the full CaSMV code was generated by SFCheck from the textual representation of the SFCs without any user interaction. This is, however, not always possible since the control programs might use language fragments which are not implemented or where additional abstractions are needed. Moreover, the verification goals have to be set by the user which requires a certain proficiency in temporal logics. The same holds for the interpretation of the verification results. We illustrate this for selected system properties.

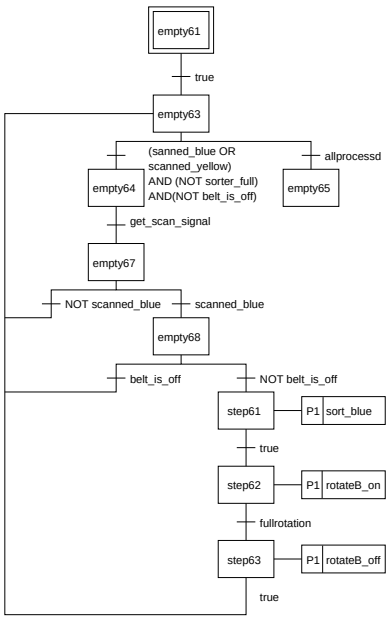

Figure 6.3: Sub-SFC of the control process

Some Verification Properties

Mutual exclusion of sorter motors. One obvious requirement for the brick sorting software is that the motors B and C never run simultaneously, i.e., each brick is sorted uniquely to one group. This requirement can be stated in the linear time temporal logic LTL as follows:

$$\Box\neg(activeA\_rotateB\_on \wedge activeA\_rotateC\_on).$$

This formula reads as follows: For all possible executions the actions *rotateB_on* and *rotateC_on* are never active at the same moment. However, checking this requirement with CaSMV yields that the property does in fact not hold and a counter trace is produced. From analyzing this counter trace it can directly be derived that the proof of this property needs an additional assumption, namely, blue and yellow are never scanned at the same time. This means, the scanning device has to produces a unique result. This leads to the formula

$$\begin{array}{rl} & \Box\neg(scanned\_yellow \wedge scanned\_blue) \\ \rightarrow & \Box\neg(activeA\_rotateB\_on \wedge activeA\_rotateC\_on) \end{array}$$

which can be proven. The CaSMV code for this property is very similar to the logic notation above where $\neg$ is replaced by `~` and $\Box$ by `G` (globally):

```
assert ((G ~(scanned_yellow & scanned_blue)
    ->  (G ~(activeA_rotateB_on & activeA_rotateC_on)));
```

Note that the scanner is working correctly, i.e., it always detects either yellow or blue in the right way, cannot be proven on the abstract level of the control SFC but has to be shown on a lower level or even on hardware level.

Liveness of sorting process. Another question that arises is whether it always is guaranteed that a brick will be sorted if scanned correctly. This property is expressed in LTL as

$$\begin{array}{rl} & \Box(scanned\_blue \vee scanned\_yellow) \\ \rightarrow & \Diamond(activeA\_rotateB\_on \vee activeA\_rotateC\_on) \end{array}$$

and means that for all executions whenever blue or yellow is scanned, eventually one of the sorters will be activated. The CaSMV notation is similar to this one where the eventually operator $\Diamond$ is replaced by `F` (finally):

```
assert  G (scanned_blue | scanned_yellow)
         -> F (activeA_rotateB_on | activeA_rotateC_on);
```

From Figure 6.3 it is already clear that the single assumption that blue or yellow is scanned is not sufficient. There are several other requirements that have to be fulfilled as well: There has to be a scan signal, the signal has to be stable, e.g., scanning blue will persist for some cycles, the belt has to be running and other more implicit requirements. However, after adding all these assumptions this property can be proven as well. The information, which requirements have to be fulfilled, can be obtained from the analysis of the counter traces produced by the model checker.

Verification Results

The above proofs of properties as well as several were computed on a SUN SPARC Ultra I with 1 GB RAM. The verification time for each property varies from less than a second to some twenty seconds. In general, liveness properties, i.e., properties with no finite counter examples (if any) take longer to compute than safety properties. For all computations the full control SFC consisting of seven parallel processes was taken into account. The abstract system consists of 56 Boolean state variables which leads to potentially 2^{56} different states.

More interesting than these raw numbers is the information gathered about the system during the verification process. E.g., the requirements that the scanning device has to return some unique information or the scanning signal has to be stable. This proved to be valuable also in validating the system and debugging malfunctions. Moreover, counter traces produced by the model checker did sometimes highlight specific requirements of the plant layout as well. E.g., the control program works only correct if the sorter is right after the scanning device. Otherwise, we were able to prove some undesired behavior. Hence, the program does not work correctly for any system layout. This has to be taken into account when extending or re-designing the plant.

6.3 A Control Trigger

A real-life industrial case study has been provided to us by a leading chemical company. It describes a measuring process where depending on the sampling subsequent actions are triggered. For reasons of non-disclosure we do not go into any technical details but briefly comment on some results.

For the case study a controlling SFC with 39 steps, 16 different actions and 20 different guards was given. After minor adaptions and abstractions (mainly replacing complex guards involving computations by Boolean variables) to fit it to our framework, the whole SFC was translated by SFCheck to CaSMV. The analysis times are comparable to the case study above.

Although the given requirements were met, we discovered a potential flaw. Some competing guards of alternative transitions are not disjoint and

if they are true, any of the transitions can chosen arbitrarily (i.e., according to the choices made in the development software). The correctness of the control SFC relies fully on the implicit assumption that all guards are evaluated from left to right, even though no priorities are given. While this assumption holds for the used development software, it is not true in general. This means if switching to another development software, the program is likely to fail.

For the given control SFC the potential flaw is not safety-critical, since it will only enter an error state, although there is no error. Hence, it behaves more conservative than necessary. From a financial point of view, however, this false alarm can be severe, since it might imply the shutdown of the process, error tracking and a subsequent restart.

Most notably to mention is that it appears possible to apply formal verification techniques, like model checking, for the analysis of SFCs up to industrial size. Moreover, little effort was necessary to do so.

6.4 A Batch Plant

The experimental batch plant we present in this section originates from the Chemical Engineering department at the University of Dortmund. It has been used as a case study for various aspects in the Esprit project on the Verification of Hybrid Systems.

The piping and instrumentation diagram of this plant is depicted in Figure 6.4. The plant produces batches of diluted salt solution from concentrated salt solution an water. In the beginning of the process the concentrated salt solution is stored in tank B1 and the water in tank B2. These ingredients are mixed in tank B3 to obtain the diluted solution, which is subsequently drained into tank B4 and then further on into B5. In B5 the solution is heated and an evaporation process is started. The evaporated water is cooled down in the condenser from where it goes to tank B6. In B6 it is further cooled down and then pumped back to $B2$. The remaining hot, concentrated salt solution in tank B5 is drained to tank B7. There is cools down to a certain temperature is reached and subsequently pumped to B1.

All operations are closely guided by different types of sensors: Liquid level sensors (LIS), temperature sensors (TI), sensors for measuring the concentration of the salt solution (QIS) and the pumps status (PIS). The whole controlling process is determined by the input from the different sensors and decisions are taken accordingly.

The whole plant is PLC driven. The code we analyzed is completely written in IL and comprises over $1,500$ lines of code. Moreover, 113 input, output and program variables are used. The program performs mainly simple comparison and decision procedures. There are about 110 jump operations to about 55 labels.

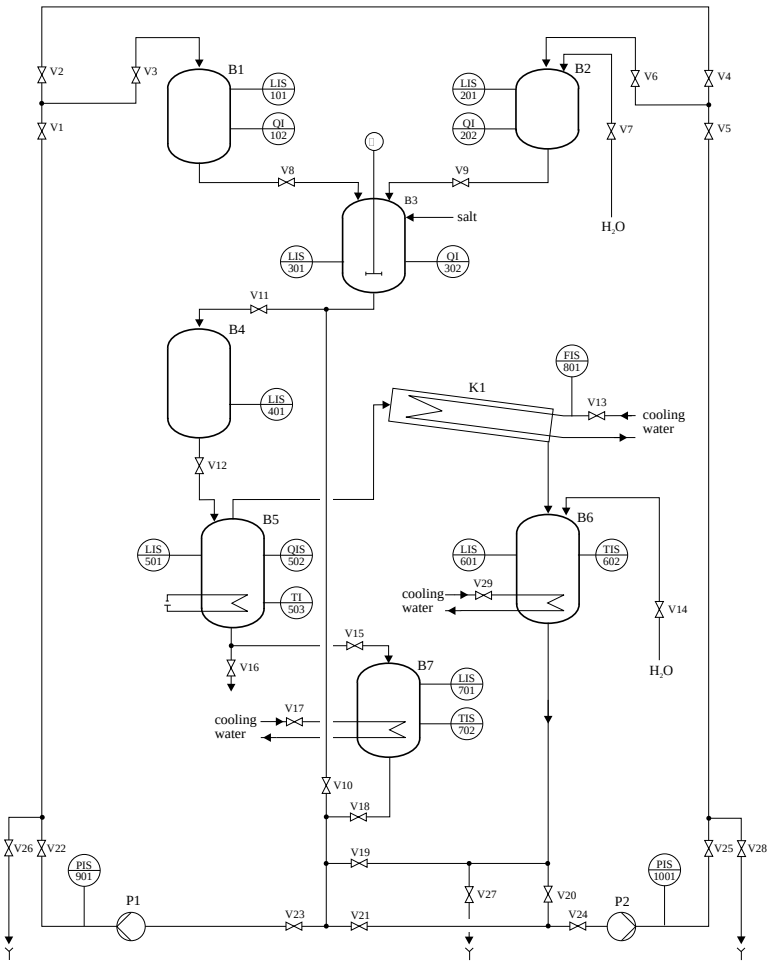

Figure 6.4: Chemical plant layout

The analysis time of the program by Homer takes about 16 seconds on Intel Pentium III mobile CPU with 1000 MHz. Note that Homer is written with a focus on clear programming and easy tracking down of analysis results There is a lot of of room for improvement and optimization. The exact execution times for this case study are listed below.

type	seconds
real	16.109
user	15.750
sys	0.140

Mainly, the analysis revealed some dead code. Moreover, there are a number of unused variables, which indicates that either some variables where declared but not needed or that other variables where used when instead some of the undeclared ones should have been used. One reason for this might be that a lot of variable names are close to each other which resulted in typos and double usage from cut-and-past operations.

The abstract interpretation process for this case study worked well, but could not find any new errors. The reason is that the program has been tested and used for the plant for quite some time and appears now to be correct. Moreover, it performs relatively simple and straightforward control decisions, where errors are not likely to occur. A small extract is shown in Figure 6.5 which illustrates the nature of this program.

```
Start1_1:
  LDN P1_1_X
  ANDN T1_1
  JMPC Start1_2

  LDN P1_1_X
  AND T1_1
  JMPCN L1_1

  LD true
  ST P1_1_X
  ST V8
```

Figure 6.5: Sample of the IL control program

We added some additional flaws to the program by purpose. For instance, assumed that some input variables are always $true$ or always $false$. As a result Homer revealed several invariant jumps, unreachable code etc. Moreover, we tested Homer on various smaller case studies we developed mainly for testing purposes. When we inserted obvious bugs, such as constants leading to invariant jumps etc., Homer performed well. However,

if the invariant jumps resulted from more complex computations involving loops, the analysis was often imprecise leading to many false alarms. This was as expected without the use of precision improvement techniques.

An additional constraint solving as suggested in Section 5.3.4 was able to reduce the number of false alarms significantly. This is, however, not yet integrated into Homer. Future improvements should, therefore, concentrate on this integrations and experiments with different lengths for the introduced history expressions as well as the use of relational data abstractions such as polyhedra instead of intervals.

Chapter 7

Conclusion

7.1 Summary

In this thesis we presented combined approaches to software verification of PLC languages. For two chosen PLC languages, SFCs and IL, we defined a formal operational semantics and applied various verification techniques.

The operational semantics defined for SFCs is the most comprehensive on for SFCs existing today. It comprises SFC features such as hierarchy, parallelism, priorities on transitions, as well as actions, and action qualifiers, which have been covered only partially before. Moreover, we sketched an extension of the semantics to timed SFCs which led to linear hybrid systems. For a significant sub-language of IL we defined a structural operational semantics. This is the first comprehensive semantics developed for IL.

According to the different natures of these two programming languages we used different verification approaches. The main approach for SFCs is model checking. This has been implemented by the development of a translation from SFCs to the input language of the model checker CaSMV, which allows to analyze temporal logic properties of SFCs. Moreover, we gave a characterization of safe SFCs, i.e., a subset of SFCs, which comply to several well-formedness requirements exceeding the actual syntax. To check for safe SFCs we modified the translation from SFCs to CaSMV and generated proof obligations, which can be automatically checked. These techniques have been implemented in a tool SFCheck, based on the standard notation for SFCs as in IEC 61131-3, and a tool S7check, based on a Siemens notation.

For IL we developed and a number of combined analysis and verification techniques. We defined an abstract simulation of IL programs, based on an abstract semantics, that approximates program behavior for sets (intervals) of possible input values. Moreover, we developed an abstract interpretation framework for IL, which is in particular helpful for range and type checking

IL programs. A heuristics for improving the precision of the abstract interpretation process was presented. We combined this framework with various data flow analysis techniques to allow further analysis of generic properties for IL programs. The combination of both techniques can significantly enhance the traditional data flow analysis, e.g., in checking for unreachable code or infinite loops. These techniques have been implemented into a tool called Homer.

The techniques and tools have been applied to a number of case studies both from academia and industry. They proved to be well capable of analyzing even large scale PLC software.

7.2 Lessons Learned

In the following we sum up some observations that resulted from this work.

Understanding. Naturally, solving a problem requires to understand the problem first. With respect to this, a formal semantics for SFCs and IL proved highly valuable for the latter analysis process. In particular the SFC language as defined in the standard is full of ambiguities and incomplete descriptions which was remedied through a formal semantics. This semantics served as a clear basis for further discussion.

Model Checking. One fundamental disadvantage of model checking results from the state explosion problem, which states a worst case exponential blow up by the number of components involved. In our point of view, however, (discrete) model checking techniques are mature enough to cope with considerably large systems such as the abstract SFCs in this work. Considering just the graph structure of the SFCs (including actions etc.) only a fraction of the model checker's potential was used, although some SFCs were probably at the limit of what is still readable by a programmer.

Abstract Interpretation. Although the idea of abstract interpretation stems from the late 1970s, comparatively few applications have found their way into static program analysis. We see, however, great potential, particularly in the combination with other static analysis techniques. The precision of the analysis is crucial for the abstract interpretation process and we believe there is still a lot of room for improvements.

Combinations. As already reasoned above, we postulate that different formal verification techniques should be used according to the problem to be solved and the type of system to be analyzed. E.g., while model checking makes sense for abstract SFCs, it does not for the type of properties we had in mind for IL. Combining different methods such

as abstract interpretation and data flow analysis can lead to results more precise than by each of them alone. In this context we believe that many formal methods can serve as a heuristics for other ones to improve the overall combined analysis results.

7.3 Future Work

There are a couple of future challenges. One of them is the extension to the verification of timed SFCs. If model checking is used for timed SFCs, the major obstacle is the increasing state space complexity introduced by timers, in particular if one considers linear hybrid systems. There are several possible ways to cope with this: By neglecting hierarchical SFCs the problem can be reduced from coping with timed systems instead of linear hybrid systems. It is likely, however, that additional techniques are still necessary. This can lead either to further abstractions or to decomposition methods [Dij69a], e.g., based on assumption/commitment techniques as suggested by [Jon81, Jon83, MC81] and already applied by us in the context of hybrid systems [HLFE02].

Future work for IL should also include an extension of the current approach to time and timers. Moreover, we see potential in developing more sophisticated heuristics to improve the precision of the abstract interpretation process. This can lead to the use of more complex data types than, e.g., intervals, or further improvements of the concept of history expressions.

An extension to the remaining languages of IEC 61131-3 should be pursued. We propose the use of abstract interpretation for structured text and model checking for ladder diagrams. For the function block language further investigations are necessary to come up with appropriate verification techniques. One solution could be a combination of control flow analysis and model checking.

Since PLC programming allows the mixture of different programming languages, e.g., SFCs whose guards are written in ladder diagram, the ultimate goal should be the integration of the different programming languages and verification techniques.

Appendix A

Chemical Plant Code

This appendix presents the original SFC as well as the CaSMV code generated by SFCheck used for the chemical plant presented in Section 5.2.2.

A.1 SFC Code

The SFC code is given in its textual representation. It should be clear from the context.

```
INITIAL_STEP s0: END_STEP

STEP s1   :           END_STEP
STEP s2   : a1 (N);   END_STEP
STEP s3   :           END_STEP
STEP s4   :           END_STEP
STEP s5   : a2 (N);   END_STEP
STEP s6   :           END_STEP
STEP s7   : a3 (N);   END_STEP
STEP s8   :           END_STEP

TRANSITION FROM s0 TO (s1, s4, s7) start       END_TRANSITION
TRANSITION FROM s1 TO s2           (NOT s11.X) END_TRANSITION
TRANSITION FROM s2 TO s3           LISplus1    END_TRANSITION
TRANSITION FROM s4 TO s5           (NOT s12.X) END_TRANSITION
TRANSITION FROM s5 TO s6           LISplus2    END_TRANSITION
TRANSITION FROM s7 TO s8           finished    END_TRANSITION
TRANSITION FROM (s3, s6, s8) TO s0 TRUE       END_TRANSITION

ACTION a1: V1  END_ACTION
ACTION a2: V2  END_ACTION
ACTION a3:

       (*********** sub SFC starting here ***********)

       INITIAL_STEP s10: END_STEP

       STEP s11   : a4 (N); a5 (S);           END_STEP
       STEP s12   : a6 (N);                   END_STEP
       STEP s13   : a7 (N); a5 (R); a8 (P0);  END_STEP
```

```
      TRANSITION FROM s10 TO s11  LIS1plus AND LISminus3 END_TRANSITION
      TRANSITION FROM s11 TO s12  LISminus1      END_TRANSITION
      TRANSITION FROM s12 TO s13  LISminus2      END_TRANSITION
      TRANSITION FROM s13 TO s10  LISminus3      END_TRANSITION

      ACTION a4: V3  END_ACTION
      ACTION a5: M   END_ACTION
      ACTION a6: V4  END_ACTION
      ACTION a7: V5  END_ACTION
      ACTION a8: finished END_ACTION

      (*********** sub SFC ending here ***********)

END_ACTION
```

A.2 CaSMV Code

The CaSMV code of the previous SFC has been generated automatically by SFCheck. Its notations are explained in Section 5.2.2.

```
  module main()

/* the following variables are detected automatically */
/* they will be considered as uninitialized Booleans  */
   start : boolean;
   LISplus1 : boolean;
   LISplus2 : boolean;
   finished : boolean;
   LIS1plus : boolean;
   LISminus1 : boolean;
   LISminus2 : boolean;
   LISminus3 : boolean;

/* declaration of ready steps */
   readyS_s0 : boolean;
   readyS_s1 : boolean;
   readyS_s2 : boolean;
   readyS_s3 : boolean;
   readyS_s4 : boolean;
   readyS_s5 : boolean;
   readyS_s6 : boolean;
   readyS_s7 : boolean;
   readyS_s8 : boolean;

/* declaration of active actions */
   activeA_a1 : boolean;
   activeA_a2 : boolean;
   activeA_a3 : boolean;

/* declaration of stored actions */

/* initialization ready steps */
   init(readyS_s0) := 1;
   init(readyS_s1) := 0;
   init(readyS_s2) := 0;
   init(readyS_s3) := 0;
   init(readyS_s4) := 0;
   init(readyS_s5) := 0;
```

```
   init(readyS_s6) := 0;
   init(readyS_s7) := 0;
   init(readyS_s8) := 0;

/* initialization of active actions */
   init(activeA_a1) := 0;
   init(activeA_a2) := 0;
   init(activeA_a3) := 0;


/* transition relation on ready steps */

default next(readyS_s0) := readyS_s0;
in case
   (readyS_s3 & readyS_s6 & readyS_s8 & next(TRUE)) : next(readyS_s0) := 1;
   (readyS_s0 & next(start)) : next(readyS_s0) := 0;


default next(readyS_s1) := readyS_s1;
in case
   (readyS_s0 & next(start)) : next(readyS_s1) := 1;
   (readyS_s1 & (~(readyS_s11 & activeA_a3))) : next(readyS_s1) := 0;


default next(readyS_s2) := readyS_s2;
in case
   (readyS_s1 & (~(readyS_s11 & activeA_a3))) : next(readyS_s2) := 1;
   (readyS_s2 & next(LISplus1)) : next(readyS_s2) := 0;


default next(readyS_s3) := readyS_s3;
in case
   (readyS_s2 & next(LISplus1)) : next(readyS_s3) := 1;
   (readyS_s3 & readyS_s6 & readyS_s8 & next(TRUE)) : next(readyS_s3) := 0;


default next(readyS_s4) := readyS_s4;
in case
   (readyS_s0 & next(start)) : next(readyS_s4) := 1;
   (readyS_s4 & (~(readyS_s12 & activeA_a3))) : next(readyS_s4) := 0;


default next(readyS_s5) := readyS_s5;
in case
   (readyS_s4 & (~(readyS_s12 & activeA_a3))) : next(readyS_s5) := 1;
   (readyS_s5 & next(LISplus2)) : next(readyS_s5) := 0;


default next(readyS_s6) := readyS_s6;
in case
   (readyS_s5 & next(LISplus2)) : next(readyS_s6) := 1;
   (readyS_s3 & readyS_s6 & readyS_s8 & next(TRUE)) : next(readyS_s6) := 0;


default next(readyS_s7) := readyS_s7;
in case
   (readyS_s0 & next(start)) : next(readyS_s7) := 1;
   (readyS_s7 & next(finished)) : next(readyS_s7) := 0;


default next(readyS_s8) := readyS_s8;
in case
```

```
   (readyS_s7 & next(finished)) : next(readyS_s8) := 1;
   (readyS_s3 & readyS_s6 & readyS_s8 & next(TRUE)) : next(readyS_s8) := 0;



/* transition relation on active actions */

default next(activeA_a1) := 0;
in case
   (next(readyS_s2)) : next(activeA_a1) := 1;


default next(activeA_a2) := 0;
in case
   (next(readyS_s5)) : next(activeA_a2) := 1;


default next(activeA_a3) := 0;
in case
   (next(readyS_s7)) : next(activeA_a3) := 1;



/* translation of sub SFC a3 */

/* declaration of ready steps */
   readyS_s10 : boolean;
   readyS_s11 : boolean;
   readyS_s12 : boolean;
   readyS_s13 : boolean;

/* declaration of active actions */
   activeA_a4 : boolean;
   activeA_a5 : boolean;
   activeA_a6 : boolean;
   activeA_a7 : boolean;
   activeA_a8 : boolean;

/* declaration of stored actions */
   storedA_a5 : boolean;

/* initialization ready steps */
   init(readyS_s10) := 1;
   init(readyS_s11) := 0;
   init(readyS_s12) := 0;
   init(readyS_s13) := 0;

/* initialization of active actions */
   init(activeA_a4) := 0;
   init(activeA_a5) := 0;
   init(activeA_a6) := 0;
   init(activeA_a7) := 0;
   init(activeA_a8) := 0;

/* initialization of stored actions */
   init(storedA_a5) := 0;



/* transition relation on ready steps */

default next(readyS_s10) := readyS_s10;
```

```
in case
   (readyS_s13 & next(LISminus3) & activeA_a3) : next(readyS_s10) := 1;
   (readyS_s10 & next(LIS1plus) & next(LISminus3) & activeA_a3) : next(readyS_s10) := 0;


default next(readyS_s11) := readyS_s11;
in case
   (readyS_s10 & next(LIS1plus) & next(LISminus3) & activeA_a3) : next(readyS_s11) := 1;
   (readyS_s11 & next(LISminus1) & activeA_a3) : next(readyS_s11) := 0;


default next(readyS_s12) := readyS_s12;
in case
   (readyS_s11 & next(LISminus1) & activeA_a3) : next(readyS_s12) := 1;
   (readyS_s12 & next(LISminus2) & activeA_a3) : next(readyS_s12) := 0;


default next(readyS_s13) := readyS_s13;
in case
   (readyS_s12 & next(LISminus2) & activeA_a3) : next(readyS_s13) := 1;
   (readyS_s13 & next(LISminus3) & activeA_a3) : next(readyS_s13) := 0;




/* transition relation on active actions */

default next(activeA_a4) := 0;
in case
   (next(readyS_s11)) & next(activeA_a3) : next(activeA_a4) := 1;


default next(activeA_a5) := 0;
in case
   (next(readyS_s13)) & next(activeA_a3) : next(activeA_a5) := 0;
   ((next(readyS_s11)) & next(activeA_a3)) | next(storedA_a5) : next(activeA_a5) := 1;


default next(activeA_a6) := 0;
in case
   (next(readyS_s12)) & next(activeA_a3) : next(activeA_a6) := 1;


default next(activeA_a7) := 0;
in case
   (next(readyS_s13)) & next(activeA_a3) : next(activeA_a7) := 1;


default next(activeA_a8) := 0;
in case
   ((readyS_s13 & ~next(readyS_s13))) : next(activeA_a8) := 1;



/* transition relation on stored actions */

default next(storedA_a5) := storedA_a5;
in case
   (next(readyS_s13)) & next(activeA_a3) : next(storedA_a5) := 0;
   (next(readyS_s11)) & next(activeA_a3) : next(storedA_a5) := 1;
```

Bibliography

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[ACHH93] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. H. Ho. Hybrid automata: An algorithmit approach to the specification and verification of hybrid systems. In *Hybrid Systems*, volume 736 of *LNCS*, pages 209–229. Springer-Verlag, 1993.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AFS98] A. Aiken, M. Fähndrich, and Z. Su. Detecting races in Relay Ladder programs. In *Proceddings of TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, pages 184–199. Springer-Verlag, 1998.

[AHLP00] Rajeev Alur, Tom Henzinger, Gerardo Lafferriere, and George J. Pappas. Discrete abstractions of hybrid systems. volume 88, pages 971–984, July 2000.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[AT98] S. Anderson and K. Tourlas. Design for proof: An approach to the design of domain-specific languages. volume 10, pages 452–468, 1998.

[aut99] *Special Issue on Hybrid Systems*, volume 35 of *Automatica*, 1999.

[Bau98] N. Bauer. Übersetzung von Steuerungsprogrammen in formale Modelle. Master's thesis, University of Dortmund, 1998.

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14:25–59, 1987.

[BCG95] Girish Bhat, Rance Cleaveland, and Orna Grumberg. Efficient on-the-fly model checking for CTL*. In *LICS '95: 10th Annual IEEE Symposium on Logic in Computer Science, San Diego, California, USA, June 26–29, 1995*, pages 388–397. IEEE Computer Society Press, 1995.

[Bel57] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[BG92] Gerard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BGL+00] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Mu noz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center.

[BH01] N. Bauer and Ralf Huuck. Towards automatic verification of embedded control software. In *Asian Pacific Conference on Quality Software*, IEEE, December 2001.

[BH02] N. Bauer and Ralf Huuck. A parameterized semantics for sequential function charts. In *Workshop of Semantic Foundations of Engineering Design Languages (SFEDL)*, April 2002. Satellite Event of ETAPS 2002.

[BHL02] N. Bauer, Ralf Huuck, and Ben Lukoschus. A stopwatch semantics for hybrid controllers. In *b'02: The XV. IFAC World Congress*, IFAC, July 2002. to appear.

[BHLL00a] Sébastien Bornot, Ralf Huuck, Yassine Lakhnech, and Ben Lukoschus. An abstract model for sequential function charts. In *Discrete Event Systems: Analysis and Control, Proceedings of WODES 2000: 5th Workshop on Discrete Event Systems, Ghent, Belgium, August 21–23, 2000*, The Kluwer International Series in Engineering and Computer Science, pages 255–264, 2000.

[BHLL00b] Sébastien Bornot, Ralf Huuck, Yassine Lakhnech, and Ben Lukoschus. Utilizing static analysis for programmable logic controllers. In *ADPM 2000: 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems, September 18-19, 2000, Dortmund, Germany*, 2000.

[BHR84] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Communications of the ACM*, 31(3):560–599, 1984.

[BKSL00] N. Bauer, S. Kowalewski, G. Sand, and T. Löhl. Case study: A demonstration plant for the control and scheduling of multi-product batch. In *ADPM2000 Conference Proceedings*, pages 383–388. Shaker Verlag, 2000.

[BKT] N. Bauer, S. Kowalewski, and H. Treseler. Model checking of control software under consideration of the PLC behaviour. Submitted to the 5th Workshop on Discrete Event Systems.

[BLJ91] A. Benveniste, P. Leguernic, and Ch. Jacquemont. Synchronous programming with events and relations. *Science of Computer Programming*, 16:103–149, 1991.

[BMR02] Thomas Ball, Todd Millstein, and Sriram K. Rajamani. Polymorphic predicate abstraction. Technical Report MSR-TR-2001-10, Microsoft Research, June 2002.

[BMS99] F. Bonfatti, P.D. Monari, and U. Sampieri. *IEC 1131-3 Programming Methodology*. CJ International, Fontaine, France, first edition, 1999.

[Bou92] François Bourdoncle. *Sémantiques des Langages Impératifs d'Ordre Supérieur et Interprétation Abstraite*. PhD thesis, École Polytechnique, 1992.

[Bou93] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the International Conference on Formal Methods in Programming and their Applications*, volume 735 of *LNCS*, pages 128–141, 1993.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Principles of Programming Languages*, January 2002.

[Bri67] G. Brinkhoff. *Lattice Theory*. American Mathematics Society, Providence, RI, 3rd edition, 1967.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992. Preprint version published as CMU Technical Report CMU-CS-92-160.

[BT01] N. Bauer and Heinz Treseler. Vergleich der Semantik der Ablaufsprache nach IEC 61131-3 in unterschiedlichen Programmierwerkzeugen. GMA Kongress, 2001, Baden-Baden, Germany, 2001.

[Cas01] Paul Caspi. Embedded control: From asynchrony to synchrony and back. In *1st International Workshop on Embedded Software, EMSOFT2001*, volume 2211 of *LNCS*, October 2001.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[CCL+00] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and Ph. Schnoebelen. Towards the automatic verification of PLC programs written in Instruction List. In *Proc. IEEE Int. Conf. Systems, Man and Cybernetics (SMC'2000)*, pages 2449–2454, 2000.

[CDH+00] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, June 2000.

[CDH01] James Corbett, Matthew Dwyer, and John Hatcliff. Expressing checkable properties of dynamic systems: The Bandera specification language. Technical Report 2001-04, KSU CIS, June 2001.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons for branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs Workshop, IBM Watson Research Center, Yorktown Heights, New York, May 1981*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1982.

[CF95] Tongwen Chen and Bruce Francis. *Optimal Sampled-Data Control Systems*. Springer-Verlag, 1995.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[CMP00] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, April 2000. ISBN : 2-84177-121-0.

[Cou78] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes*. PhD thesis, Université scientifique et médicale de Grenoble, France, 1978.

[Cou81] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[Cou01] P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.

[CVWY92] Constantin Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[DA92] R. David and H. Alla. *Petri Nets & Grafcet*. Prentice Hall, 1992.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *ACM SIGPLAN Notices*, 29(6):230–241, 1994.

[DFMV98] H. Dierks, A. Fehnker, A. Mader, and F. Vaandrager. Operational and logical semantics for polling real-time systems. Technical report, University of Nijmegen, 1998.

[DH99] Matthew B. Dwyer and John Hatcliff. Slicing software for model construction. In *ACM Workshop on Partial Evaluation and Semantic-Based Program Manipulation*, pages 105–118, 1999.

[DHJ+01] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Pasareanu, Robby, Willem Visser, and Hongjun Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.

[Die97a] H. Dierks. PLC-automata: A new class of implementable real-time automata. In M. Bertran and T. Rus, editors, *Transformation-Based Reactive Systems Development (ARTS'97)*, volume 1231 of *Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, 1997.

[Die97b] H. Dierks. Synthesising controllers from real-time specifications. In *Proceedings of Tenth International Symposium on System Synthesis*, pages 126–133. IEEE CS Press, 1997.

[Dij69a] Edsger W. Dijkstra. On understanding programs (EWD 264). Published in an extended version as [Dij69b], August 1969.

[Dij69b] Edsger W. Dijkstra. Structured programming. In J.N. Buxton and B. Randell, editors, *Software Engineering Techniques, Report on a conference sponsored by the NATO Science Committee*, pages 84–88. NATO Science Committee, 1969.

[Dil90] David Dill. Timing assumptions and verification of finite-state concurrent systems. In Joseph Sifakis, editor, *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France, June 12–14, 1989*, volume 407 of *LNCS*, pages 197–212. Springer-Verlag, 1990.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report #159, Palo Alto, USA, 1998.

[DM98] T. Dang and O. Maler. Reachability analysis via face lifting. In *Hybrid Systems Computation and Control*, volume 1386 of *LNCS*, pages 96–109, 1998.

[DN00] J. Davoren and A. Nerode. Logics for hybrid systems. *Proceedings of the IEEE*, 88, July 2000.

[DP90] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, 1990.

[Edw97] Stephen A. Edwards. *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*. PhD thesis, University of California, Berkley, 1997.

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[Eva96] David Evans. Static detection of dynamic memory errors. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, 1996.

[fac] FaCiLe: A functional constraint library. `http://www.recherche.enac.fr/opti/facile/`.

[fea] The feaver feature verification system. `http://cm.bell-labs.com/cm/cs/what/feaver/`.

[FHL+01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Rheinhard Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Embedded Software Workshop*, Lake Tahoe, USA, October 2001. Springer-Verlag.

[FK92] A. Falcione and B.H. Krogh. Design recovery for relay ladder logic. In *First IEEE Conference on Control Applications*, volume 2 of *IEEE*, pages 648–653, September 1992.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245. ACM Press, 2002.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Proceedings AMS Symposium Applied Mathematics*, volume 19, pages 19–31, Providence, RI, 1967. American Mathematical Society.

[Gra89] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30:165–190, 1989.

[Gra91] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT '91: Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 493 of *LNCS*, pages 169–192, 1991.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, 1977.

[HFL01] Anders Hellgren, Martin Fabian, and Bengt Lennartson. On the execution of discrete event systems as sequential function charts. In *Conference on Control Applications*, IEEE, September 2001.

[HHWT97] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1:110–122, 1997.

[HKPV98] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaija. What's decidable about hybrid automata? *Journal of Comput. Syst. Sci.*, 57:94–124, 1998.

[HLFE02] Ralf Huuck, Ben Lukoschus, Goran Frehse, and Sebastian Engell. Compositional verification of continuous-discrete systems. In S. Engell, G. Frehse, and E. Schnieder, editors, *Modelling, Analysis and Design of Hybrid Systems*, volume 279 of *Lecture Notes in Control and Information Sciences*, pages 225–244. Springer-Verlag, 2002.

[HM98] M. Heiner and T. Menzel. A Petri net semantics for the PLC language Instruction List. In *Proceedings of the International Workshop on Discrete Event Systems (WoDES)*, pages 161–166. IEE Control, 1998.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Engelwood Cliffs, 1985.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[Hol02] G.J. Holzmann. Static source code checking for user-defined properties. Pasadena, CA, USA, June 2002.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. pages 447–498, 1985.

[HP00] Klaus Havelund and Tom Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, April 2000. Special issue containing selected submissions for the 4'th SPIN workshop, Paris, November 1998.

[HS00] G. J. Holzmann and Margaret H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, - 2000.

[HTLW97] H.-M. Hanisch, J. Thieme, A. Lüder, and O. Wienhold. Modeling of PLC behaviour by means of timed net condition/event

systems. In *Proc. of IEEE Int. Symposium on Emerging Technologies and Factory Automation (EFTA '97)*, pages 361–369, 1997.

[Huu02] Ralf Huuck. Software verification for embedded systems. In *MMAR '02: The 8th IEEE International Conference on Methods and Models in Automation and Robotics, Szczecin, Poland, September 2–5 , 2002*, September 2002.

[IEC92] International Electrotechnical Commission, Technical Committee No. 848. *IEC 60848, Preparation of function charts for control systems*, 1992.

[IEC98] International Electrotechnical Commission, Technical Committee No. 65. *Programmable Controllers – Programming Languages, IEC 61131-3*, second edition, November 1998. Committee draft.

[iee98] *Special Issue on Hybrid Systems*, volume 43 of *IEEE Transactions of Automatic Control*, 1998.

[JFR99] F. Jimènez-Fraustro and E. Rutten. A synchronous model of the PLC programming language ST. In *1st Euromicro Conference on Real-Time Systems*, pages 21–24, June 1999.

[JM01] Ranjit Jhala and Kenneth L. McMillan. Microarchitecture verification by compositional model checking. In G. Berry and H. Comon, editors, *Computer Aided Verification 13th International Conference, CAV 2001*, volume 2102 of *LNCS*, pages 396–410, 2001.

[Jon81] Cliff B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University Computing Laboratory, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[Jon83] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[KBP+99] S. Kowalewski, N. Bauer, J. Preußig, O. Stursberg, and H. Treseler. An environment for model-checking of logic control systems with hybrid dynamics. In *Proc. IEEE Int. Symp. On Computer Aided Control System Design*, 1999.

[Kri63] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[KV97] B. J. Krämer and N. Völker. A highly dependable computer architecture for safety-critical control applications. *Real-Time Systems Journal*, 13(3):237–251, 1997.

[KVW00] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, 2000.

[Lew98] R.W. Lewis. *Programming industrial control systems using IEC 1131-3*, volume 50 of *Control Engineering Series*. The Institution of Electrical Engineers, Stevenage, United Kingdom, revised edition, 1998.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specifications. In *Twelfth ACM Symposium on the Priciples of Programming Languages*, pages 97– 105, 1985.

[LPM95] D. L'Her, P. Le Parc, and L. Marcé. Proving sequential function chart programs using automata. In *Proceedings of 2nd AMAST workshop on Real-Time Systems*, 1995.

[LPY97a] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[LPY97b] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[LT79] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. In *ACM Transactions on Progamming Languages and Systems 1*, pages 1:121–141. ACM Press, New York, NY, 1979.

[Mal97] O. Maler, editor. *Hybrid and Real-Time Systems*, volume 1201 of *LNCS*. Springer-Verlag, 1997.

[Mal99] Oded Maler. On the programming of industrial computers. Technical report, Verimag, 1999.

[Mar92] Florence Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *International Conference on Concurrency Theory*, pages 550–564, 1992.

[Mas92] François Masdupuy. Array abstractions using semantic analysis of trapezoid congruences. In *ICS '92: Proceedings of the 6th ACM International Conference on Supercomputing*, ACM, pages 226–235, 1992.

[mat] Mathlab and Simulimk. `http://www.mathworks.com`.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.

[McM92] Kenneth L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, May 1992. CMU Technical Report CMU-CS-92-131.

[McM00] Kenneth L. McMillan. *The SMV system*. Carnegie Mellon University, November 2000. Manual for SMV version 2.5.4.

[Mer01] Stephan Merz. Model checking: A tutorial overview. In F. Cassez et al., editor, *Modeling and Verification of Parallel Processes*, volume 2067 of *LNCS*, pages 3–38. Springer-Verlag, 2001.

[MH02] Angelika Mader and Ralf Huuck. Modelling methods for PLC applications. Unpublished Manuscript, 2002.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, 1980.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall International, Engelwood Cliffs, 1989.

[Moo94] I. Moon. Modeling programmable logic controllers for logic verification. *IEEE Control Systems Magazine*, 14(2):53–59, 1994.

[Mor] Dick Morley. The history of the PLC. `http://www.barn.org/FILES/historyofplc.html`.

[Muc97] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[MW99] A. Mader and H. Wupper. Timed automaton models for simple programmable logic controllers. In *Proceedings of the 11th Euromicro Conference on Real Time Systems*, pages 114–122. IEEE Computer Society, 1999.

[Nis00] Norman Nise. *Control Systems Engineering*. John-Wiley & Sons, Inc., third edition, 2000.

[NK00] N.Lynch and B. H. Krogh, editors. *Hybrid Systems: Computation and Control*, volume 1790 of *LNCS*. Springer-Verlag, 2000.

[NNH99] F. Nielson, H. Riis Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[NSY93] Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. From ATP to timed graphs and hybrid systems. *Acta Informatica*, 30(2):181–202, 1993.

[Ore44] Oystein Ore. Galois connexions. *Transactions of the American Mathematical Society*, 55:493–513, 1944.

[OY93a] A. Olivero and S. Yovine. *KRONOS: A tool for Verifying Real-Time Systems. Userguide*. VERIMAG, Grenoble, France, 1993. available at: `http://www.imag.fr/VERIMAG/TEMPORISE/kronos/`.

[OY93b] A. Olivero and S. Yovine. *KRONOS: A Tool for Verifying Real-Time Systems. User's Guide and Reference Manual*. Verimag, Grenoble, France, 1993.

[pep] *PEP Homepage*. `http://theoretica.Informatik.Uni-Oldenburg.DE/~pep/`.

[Plo81] Gordon Plotkin. A structural approach to operational semantics. DAIMI FN-19, Computer Scinece Depratment, Aarhus University, 1981.

[Pnu81] Amir Pnueli. The temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[Pol02] PolySpace C verifier: Product data sheet. `http://www.polyspace.com/docs/CLeaflet.pdf`, 2002.

[PSSD00] David Y. W. Park, University Stern, Jens U. Sakkebaek, and David L. Dill. Java model checking. In *Automated Software Engineering*, pages 253–256, 2000.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proceedings of the 5th International Symposium on Programming, Turin, April 6–8, 1982*, pages 337–350. Springer-Verlag, 1982.

[Rei85] Wolfgang Reisig. *Petri Nets, An Introduction*. EATCS, Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1985.

[RK98] M. Rausch and B. Krogh. Formal verification of PLC programs. In *American Control Conference*, pages 234–238, June 1998.

[RLR99] O. Rossi, J. J. Lesage, and J. M. Roussel. Formal validation of PLC programs: a survey. In *Proceedings of European Control Conference 1999 (ECC'99)*, 1999.

[RS00] O. Rossi and Ph. Schnoebelen. Formal modelling of timed function blocks for the automatic verification of ladder diagram programs. In *ADPM 2000: 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems, September 18-19, 2000, Dortmund, Germany*, 2000.

[Sch95] Erik Schön. On the computation of fixpoints in static program analysis with an application to analysis of AKL. Master's thesis, School of Engineering Physics, Royal Institut of Technology, Stockholm, October 1995.

[Sch99] Bernd S. W. Schröder. Algorithms for the fixed point property. *Theoretical Computer Science*, 217(2):301–358, 1999.

[SDL92] Specification and Description Language SDL, blue book. CCITT Recommendation Z.100, 1992.

[Sta98] Karsten Stahl. Comparing the expressiveness of different real-time models. Master's thesis, Christian-Albrechts-University of Kiel, May 1998.

[Tap98] J. Tapken. Moby/PLC - A Design Tool for Hierarchical Real-Time Automata. In *Proceedings of FASE'98*, volume 1382 of *Lecture Notes in Computer Science*, pages 326–329. Springer-Verlag, 1998. available at: `http://theoretica.Informatik.Uni-Oldenburg.DE/~moby/`.

[Tho90] Wolfgang Thomas. Automata on infinite objects. pages 133–191, 1990.

[TN02] M. Wenzel T. Nipkow, L. C. Paulson. *A Proof Assistant for Higher-Order Logic*. Number 2283 in LNCS. Springer-Verlag, 2002.

[Tou96] K. Tourlas. Semantic analysis and design of languages for programmable logic controllers. Master's thesis, Department of Computer Science, The University of Edinburgh, 1996.

[TPP97] A. L. Turk, S. T. Probst, and G. J. Powers. Verification of real time chemical processing systems. In O. Maler, editor, *Proc. International Workshop on Hybrid and Real-Time Systems (HAR'97)*, volume 1201 of *Lecture Notes in Computer Science*, pages 259–272. Springer-Verlag, 1997.

[Var01] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th Inter-*

national Conference, TACAS 2001, volume 2031 of *lncs*, pages 1–22. springer, 2001.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[Wil99] H.X. Willems. Compact timed automata for PLC programs. Technical Report CSI-R9925:, University of Nijmegen, Computing Science Institute, 1999. `http://www.cs.kun.nl/csi/reports/info/CSI-R9925.html`.